Converging Clean Architecture with Normalized Systems

Gerco Koks

Zundert, Netherlands gerco.koks@outlook.com

*Abstract*—This paper explores the convergence between Clean Architecture (CA) and Normalized Systems (NS) principles, highlighting their synergistic potential to enhance software design and evolvability. The paper draws upon the research described in the thesis of Koks, Through an comparative analysis of design elements and principles, it demonstrate how each paradigm contributes to modular, stable and evolvable software design, and how integration of both approaches can lead to a more widely spread adoption and even a more improved software design. The study underscores the importance of modular and decoupled design, while meticulously adhering to design principles, advocating for a unified framework that leverages the strengths of both CA and NS to address contemporary challenges in software engineering.

*Index Terms*—Software; Architecture; Evolvability; Modularity; Maintainability;

# I. INTRODUCTION

In the evolving landscape of software architecture, the software development paradigms of CA and NS have emerged as pivotal in addressing the multifaceted challenges of software design, particularly in managing stability, modularity, and evolvability in order to achieve resiliency in software artifacts. 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 their indebtedness to the seminal 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 enduring relevance of fostering maintainable and adaptable software systems.

The core of this paper is an exploration of the findings from an extensive thesis on the convergence of CA and NS [1]. This research, characterized by a thorough analysis and a scholarly approach 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.

## The Artifact

A vital aspect of the thesis, and by extension, this paper, is the empirical examination of CA and NS within a software artifact. The practical application of the artifacts substantiates the theoretical constructs, effectively narrowing down the divide between theory and practice. The research substantiates the efficacy and compatibility of these methodologies through a detailed analysis of the implementation and performance of CA and NS principles in an operational software system. This empirical dimension corroborates the theoretical perspectives and offers a pragmatic example through which these architectural paradigms can be applied in modern software development contexts.

## The goal of this paper

This paper aims to present the crucial insights from the thesis, shedding light on the significant benefits and practical implications of integrating the strengths of CA and NS within the dynamic field of software development.

## Structure of this paper

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 whilst also highlighting their unified concepts. In section 3 we analyze the synergies of these principles and elements and the effect in combinatorial effects found while developing the artifacts. The limitations of the research are discussed in Section 4, and the paper concludes with the conclusion in Section 5.

# II. THEORETICAL BACKGROUND

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

## 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 Robert C. 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, Verelst, and De Bruyn [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, Verelst, and De Bruyn [7, p. 22] consider cohesion as modules that exist out of connected or interrelated parts of a hierarchical structure. On the other hand, Robert C. Martin [6, p. 118] discusses cohesion in the context of components. He attributes the three component cohesion principles as crucial to group classes or functions into cohesive components. Cohesion is a complex and dynamic process, as the level of cohesiveness might evolve as requirements change over time.
3. *Coupling:* Coupling is an essential concept in software engineering related to the degree of interdependence among software modules and components. High coupling between modules indicates the strength of their relationship, whereby a high level of coupling implies a significant degree of interdependence. Conversely, low coupling signifies a weaker relationship between modules, where modifications in one module are less likely to impact others. Although not always possible, the level of coupling between the various modules of the system should be kept to a bare minimum. Both Mannaert, Verelst, and De Bruyn [7, p. 23] and Robert C. Martin [6, p. 130] agree with the idea that modules should be coupled as loosely as possible.

## B. Normalized Systems

NS in software enginering revolves around stable and evolvable information systemsd, 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 TCPIP firewalls [9], and Business Process Modeling [10].

The NS theory emphasizes stability as a crucial property derived from the concept of Bounded Input and 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, impacting its evolvability negatively. In the following list, we will describe the design Theorems of NS, firstly presented by Mannaert and Verelst [11].

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

The goal of NS is to design evolvable software independent of the underlying technology. Nevertheless, a particular technology must be chosen when implementing the software and its components. For Object Oriented Programming Languages like Java, the following Normalized Elements have been proposed [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, Verelst, and Ven [12, p. 102]

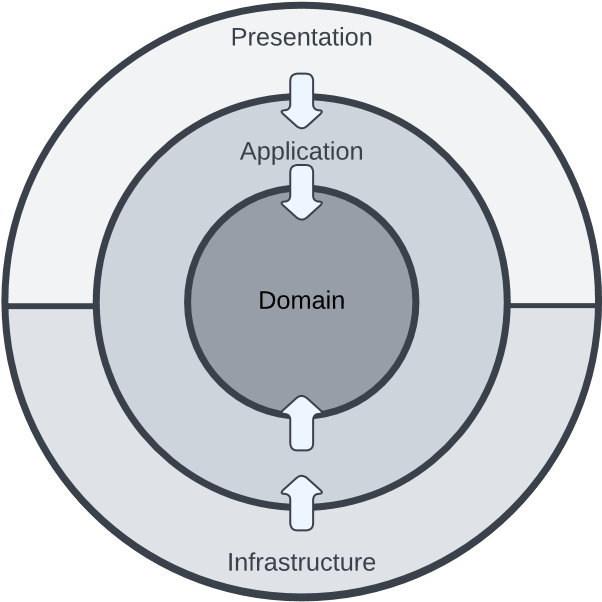
* *Data Element*: Based on Data Version Transparency (DvT), data elements have get- and set-methods for widesense data version transparency, or marshal -and parsemethods for strict-sense DvT. Supporting tasks can be added in a way which is consistent with the principles Separation Of Concerns (SoC) and DvT.
* *Task Element*: Based on SoC, the core action entity can only contain a single functional task, and not multiple tasks. Based on Action Version Transparency (AvT), arguments and parameters must be encapsulated data entities. Based on SoC and Separation of State (SoS), workflows need to be separated from action entities, and will therefore be encapsulated in a workflow element. Based on AvT, tasks need to be encapsulated in such a way that a separate action entity wraps the action entities representing task versions. Supporting tasks can be added in a way which is consistent with SoC and AvT.
* *Workflow 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 argument.
* *Action 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 which is consistent with SoC and AvT.
* *Connector Element*: Based on SoC, trigger elements need to control the separated —both error and nonerrorstates, and check whether an action element has to be triggered. Supporting tasks can be added in a way which is consistent with SoC and AvT.

## C. Clean Architecture

CA is a software design approach that emphasizes the organization of code into independent, modular layers with distinct responsibilities. This approach aims to create more flexible, maintainable, and testable software systems by enforcing the separation of concerns and minimizing dependencies between components. The goal of clean architecture is to provide a solid foundation for software development, allowing developers to build applications that can adapt to changing requirements, scale effectively, and remain resilient against the introduction of bugs [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 software systems that are more flexible, maintainable, and testable, with well-defined boundaries and 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 indepen- dent of any specific technology or framework. The domain layer focuses on encapsulating the essential complexity of the system and should be kept as pure as possible.
* *Application Layer*: This layer contains the use cases or application-specific business rules orchestrating the interaction between entities and external systems. Use cases define the application’s behavior regarding the actions users can perform and the expected outcomes. This layer is responsible for coordinating the flow of data between the domain layer and the presentation or infrastructure layers while remaining agnostic to the specifics of the user interface or external dependencies.
* *Presentation Layer*: This layer translates data and interactions between the use cases and external actors, such as users or external systems. Interface adapters include controllers, view models, presenters, and data mappers, which handle user input, format data for display, and convert data between internal and external representations. The presentation layer should be as thin as possible, focusing on the mechanics of user interaction and deferring application logic to the use cases.
* *Infrastructure Layer*: This layer contains the technical implementations of external systems and dependencies, such as databases, web services, file systems, or thirdparty 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.



### Fig. 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.

Robert C. Martin [6, p. 78] argues that software can quickly become a well-intended mess of bricks and building blocks without a rigorous set of design principles. So, from the early 1980s, he began to assemble a set of software design principles as guidelines to create software structures that tolerate change and are easy to understand. The principles are intended to promote modular and component-level software structure [6, p. 79]. In 2004 the arrangement of the principles was definitively arranged to form the acronym SOLID. The following list will provide an overview of each of the SOLID principles.

* *Single Responsibility Principle*: This principle has gone through several iterations of the formal definition. The final definition of the Single Responsibility Principle (SRP) is: *a module should be responsible to one, and only one, actor* (Robert C. Martin, 2018, p. 82). The word actor in this statement refers to all the users and stakeholders represented by the (functional) requirements. The modularity concept in this definition is described by Robert C. Martin [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. Robert C. 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*: Meyer [13] first mentioned the Open/Closed Principle (OCP) and formulated the following definition: *A module should be open for extension but closed for modification.* The software architecture should be designed such that the behavior of a module can be extended without modifying existing source code. The OCP promotes the use of abstraction and polymorphism to achieve this goal. The OCP is one of the driving forces behind the software architecture of systems making it relatively easy to apply new requirements. [6, p. 94].
* *Liskov Substitution Principle*: The Liskov Substitution Principle (LSP) is named after Barbara Liskov, who first introduced the principle in a paper she co-authored in 1987. Barbara Liskov wrote the following statement to define subtypes (Robert C. Martin, 2018, p. 95). *If for each object o1 of type S, there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.1.* Or in simpler terms: To build software from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted for another (Robert C. Martin, 2018, p. 80)
* *Interface Segregation Principle*: The Interface Segregation Principle (ISP) suggests that software components should have narrow, specific interfaces rather than broad, general-purpose ones. In addition, the ISP states that consumer code should not be allowed to depend on methods it does not use. In other words, interfaces should be designed to be as small and focused as possible, containing only the methods relevant to the consumer code using them. This allows the consumer code to use only the needed methods without being forced to implement or depend on unnecessary methods [6, p. 104].
* *Dependency Inversion Principle*: 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 that define a contract for how they should interact with each other [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 system over time, as changes can be made to specific modules without affecting the rest of the system.

Robert C. 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 are used to represent the input data required by a specific interactor.
* *ResponseModel*: ResponseModel are used to represent the output data required by a specific interactor.
* *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. THE ANALYSIS

This section delves into the convergence of CA and NS, presenting an exploration of 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 not only reinforces the robustness of software systems but also 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. An analysis of converging principles

The main goal of both Single Responsibility Principle (SRP) and SoC is to promote and encourage modularity, low coupling, and high cohesion. While their definitions has some minor nuances, the two principles are practically interchangeable. Even though SRP does not implicitly guarantee DvT or AvT, it supports those theorems by directing desing choices in a certain way. One example lies in the separation of data models for requests, responses and views and respective versions of these models.

The OCP and its relation to Normalized Systems 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 the introduction of new functionality or data elements without disrupting the core architecture, thereby supporting the OCP principle goal of extendibility and maintainability in a cohesive manner.

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. An application of this principle within NS is evident in the design of tailored interfaces specific to a particulair version, which ensures system evolution without compromising existing functionality, thereby upholding the LSP directive for substitutability and system integrity.

The ISP advocates for creating specific interfaces for consumers rather than one general-purpose interface, aligning with NS principles aimed at enhancing 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 actually use, which mirrors the approach in NS to creating 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 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 lead to an increase of 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 glsns are highly converged. Both have a similar impact on the design and implementation of the artifact.
* *Supports convergence* (+): The CA principle supports implementing the NS principle through specific design choices. However, applying the CA principle does not inherently ensure adherence to the corresponding NS principle.
* *Weak or no convergence* (−): The principles have no significant similarities in terms of their purpose, goals, or architectural supports.

Clean Architecture

NormalizedSystems

SeparationOfConcerns

DataVersionTransparency

ActionVersionTransparency

SeparationofState

Single Responsibility Principle

++ + + −

Open/Closed Principle

++ − ++ −

Liskov Substitution Principle

++ − + −

Interface Segregation Principle

++ − + −

Dependency Inversion Principle

++ − + −

TABLE I

T

HE CONVERGENCE PRINCIPLES SUMMARIZED

## B. An analysis of converging elements

Both the Data Element from NS and the Entity Element from CA represent data objects of the ontology or data schema and typically include attributes and relationship information. While both can contain a complete set of attributes and relationships, the Data Element of NS may also be tailored to serve a specific set of information required for a single task or use case. In CA, these type of Data Elements are explicitly specified as ViewModels, RequestModels, or Response Models.

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 in a statefull manner and therefore also 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, whilest the Connector and Trigger elements are intended to interact with also other type of external systems.

Both the Gateway element of CA and the Connector element of NS are involved in the communication between components by providing Data Version Transparent interfaces in order to provide Action Version Transparency between these components.

The Presenter is responsible for preparing the ViewModel on behalf of the controller, and can be therefore considdered to be a Task of Workflow Element.

The Boundary element of CA has a strong convergence with the Connector element of NS, as both are involved in communication between components and help ensure loose coupling between these components. However, the Boundary element’s scope seems 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 III-A:

Clean Architecture

NormalizedSystems

DataElements

TaskElement

FlowElement

ConnectorElement

TriggerElement

Entity Element

++−−−−

Interactor Element

− ++ ++ − −

RequestModel Element

++−−−−

ResponseModel Element

++−−−−

ViewModel Element

++−−−−

Controller Element

−−−++

Gateway Element

−−−++−

Presenter Element

−++−−

Boundary Element

−−−++−

TABLE II

T

HE CONVERGENCE ELEMENTS SUMMARIZED

## C. Analysis of Combinatorial Effects

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

Mannaert, Verelst, and De Bruyn [7, p. 137] use the term ‘Mirror world’ as an analogy that refers to the activation of the technology of the information system.

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

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

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

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

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

## D. The Craftings

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

# IV. DISCUSSION

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

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

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

# V. CONCLUSION

This paper describes a multidimensional exploration of the convergence of CA with NS, described in the Master’s thesis [1]. We have drawn upon the author’s firsthand experience designing software architectures using rigorous theoretical research. Additionally we created practical and working software artifacts. The primary objective was to study the convergence between CA and NS by analyzing their principles and design elements through theory and practice. This section will summarize the findings into a research conclusion.

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

Stability and evolvability are concepts not directly referenced in the literature on CA, but this design approach aligns with the goal of Mannaert, Verelst, and De Bruyn [7, p. 31]. As depicted in Table I, the attentive reader surely observes the shared emphasis on modularity and the separation of concerns, as all SOLID principles have a strong convergence with SoC. Both approaches attempt to achieve low coupling and high cohesion. In addition, CA adds the dimensions of dependency management as useful measures to improve maintainability and manage dependencies in a modular architecture.

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

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

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

The most notable difference between CA and NS is their approach to handling state. CA does not explicitly address state management in its principles or design elements. At the same time, NS provides the principle of Separation of State, ensuring that state changes within a software system are stable and evolvable. This principle can be crucial in developing scalable and high-performance systems, as it isolates state changes from the rest of the system, reducing the impact of state-related dependencies and side effects.

The findings can only lead to the conclusion that the convergence between CA and NS is incomplete because CA needs specific state management principles. As a result, CA cannot fully ensure stable and evolvable software artifacts as defined by NS.

## BIBLIOGRAPHY

1. G. Koks, “On the Convergence of Clean Architecture with the Normalized Systems Theorems,” en, Antwerpen Management School, Jun. 2023. [Online]. Available: https://zenodo.org/record/8029973.
2. D. McIlroy, “NATO SOFTWARE ENGINEERING CONFERENCE 1968,” 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. [Online]. Available: http: //ieeexplore.ieee.org/document/1456074/ (visited on 04/25/2022).
4. E. Dijkstra, “Letters to the editor: Go to statement considered harmful,” en, *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968, ISSN: 00010782, 1557-7317. DOI: 10 . 1145 / 362929 . 362947. [Online]. Available: https://dl.acm.org/doi/10.1145/ 362929.362947 (visited on 03/20/2023).
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, Dec. 1972, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/361598.361623. [Online]. Available: https://dl.acm.org/doi/10.1145/ 361598.361623 (visited on 03/19/2023).
6. Robert 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: 97890-77160-09-1.
8. P. Huysmans and J. Verelst, “Towards an EngineeringBased 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. [Online]. Available: http: //link.springer.com/10.1007/978-3-642-38490-5 5 (visited on 04/25/2022).

1. G. Haerens, “On the Evolvability of the TCP-IP Based Network Firewall Rule Base,” en, 2021.
2. D. van Nuffel, “Towards Designing Modular and Evolvable Business Processes,” en, p. 424, 2011.
3. H. Mannaert and J. Verelst, *Normalized systems recreating information technology based on laws for software evolvability*, English. Kermt: Koppa, 2009, OCLC: 1073467550, ISBN: 978-90-77160-00-8.
4. 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, Jan. 2012, ISSN: 00380644. DOI: 10 . 1002 / spe . 1051. [Online]. Available: https : / / onlinelibrary.wiley.com/doi/10.1002/spe.1051 (visited on 04/23/2022).
5. B. Meyer, *Object-oriented software construction*, 1st ed. Upper Saddle River, N.J: Prentice Hall PTR, 1988, ISBN: 978-0-13-629155-8.