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 [0], who championed modular programming, and Lehman [0], who underscored the importance of software evolution. Contributions from Dijkstra [0] on structured programming and Parnas [0] 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 [0]. 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 the research conducted by G. Koks, 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 [0, 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.* [0, 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.* [0, p. 22] consider cohesion as modules that exist out of connected or interrelated parts of a hierarchical structure. On the other hand, Martin [0, 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.* [0, p. 23] and Martin [0, 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

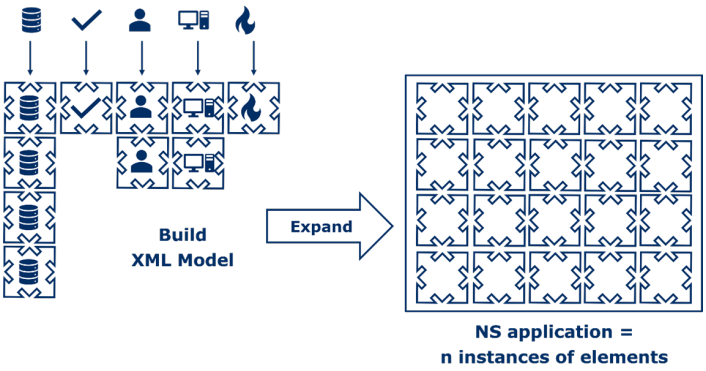

Figure 1. Requirements expressed in XML description file, used on input for element expansion.

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 (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.

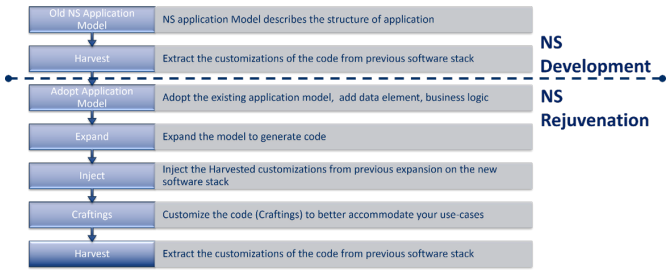

Figure 2. The NS development process.

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 [0].

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 [0]. 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.

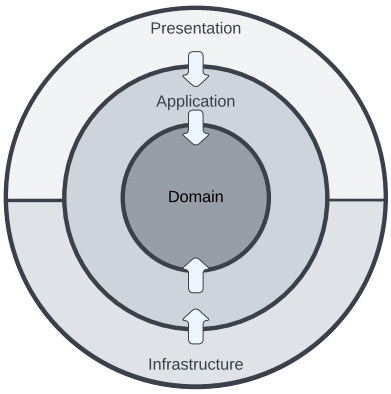

Figure 3. Flow of control

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 [0, 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 [0, 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 [0, 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 [0, 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 [0, 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 [0] 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. [0, 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 [0, 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 [0, 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 [0] 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. Comparing the principles

In this section we delves into the comparisson of the principles 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" [0], which examines the principles CA and NS mentioned in previous chapters. By aligning the theoretical constructs of both paradigms, the thesis and its artifacts provides a perspective on achieving modular, evolvable, and stable software architectures. Applying the principles of both paradigms reinforces the robustness of software systems and enhances their evolvability and longevity in the face of future requirements.

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.

TABLE I
Denotation of convergence levels.

Icon	Level	Description
++	Strong	This indicates that the principles of CA and NS are highly converged. Both have a similar impact on the design and implementation.
+	Supporting	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	he principles have no significant similarities in terms of their purpose, goals, or architectural supports.

TABLE II
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		++	−	+	−

IV. COMPARING THE ELEMENTS ELEMENTS

In this section we compare the design elements 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" [0], which examines the elements CA and NS mentioned in previous chapters, from both a theoretical and practical perspective.

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 III.

TABLE III
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		−	−	−	++	−