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 investigates Clean Architecture through the lens of Normalized Systems, expanding on the research presented in the thesis of "On the Convergence of Clean Architecture with the Normalized Systems Theorems" from G. Koks. The study highlight the synergetic potential of Clean Architecture and Normalized Systems to enhance the evolvability of Software design. The research includes a theoretical analysis, supported by imperical evidence from the development and evaluation of two research artifacts. It demonstrates how each paradigm contributes to a modular, stable, and evolvable software design and how integrating both approaches can lead an evaolvable software design.**

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

I. Introduction

In the dynamic 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 to achieve evolvability in software. This paper delves into the synergy between these two architecture '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 was one of the first to discuss modular programming, and Lehman [3], who pointed out 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.

This paper outlines the insights from a design science research conducted by G. Koks, exploring the significant benefits and practical implications of integrating the strengths of CA and NS within the field of software development [1]. Besides the theoretical study of comparing the principles and building blocks of both paradigms, the research included an architectural design artiact, and a software artifact where the principles were applied and tested in practice.

The introduction is intended to set the stage and articulate the goal of this paper. Section II lays out the theoretical background, focusing on the specific principles and elements of each Software Design Paradigm while highlighting their unified concepts. Section III delves into a detailed comparison of the principles and elements of CA and NS, examining their similarities, differences, and their impact on the evolvability of software constructs. Section IV explores the convergence of design elements between CA and NS, providing a practical perspective on their integration. Section V discusses the development and analysis of research artifacts, including the Expander Framework and Clean Architecture Expander, to evaluate the convergence of the two theories in a practical context. Section VI presents the research artifacts, detailing their construction and the methodologies used to assess their effectiveness. Finally, Section VII concludes the paper with a summary of findings, discussing the implications of the research and offering recommendations for future work in the field of software architecture.

II. Theoretical Background

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

A. Unified concepts

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

1) Modularity

The original material of Martin [6, p. 82] describes a module as a piece of code encapsulated in a source file with a cohesive set of functions and data structures. According to Mannaert *et al.* [7, p. 22], modularity is a hierarchical or recursive concept that should exhibit high cohesion. While both design approaches agree on the cohesiveness of a module's internal parts, there is a slight difference in granularity in their definitions.

2) Cohesion

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

3) Coupling

Coupling is an essential concept in software engineering that is related to the degree of interdependence among various software constructs. High coupling between components indicates the strength of their relationship, creating an interdependent relationship between them. Conversely, low coupling signifies a weaker relationship, where modifications in one part are less likely to impact others. Although not always possible, the level of coupling between the various modules of the system should be kept to a bare minimum. Both Mannaert *et al.* [7, p. 23] and Martin [6, p. 130] agree to achieve as much decoupling as possible.

B. Fundamentals of NS theory

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

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

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

1) NS Theorems

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

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

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

2) NS Elements

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

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

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

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

3) Element Expansion

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

The application's business logic is now manually programmed inside the expanded modules at pre-defined locations. The result is an application that implements a certain required business logic and has a fine-grained modular struc-

ture. 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.

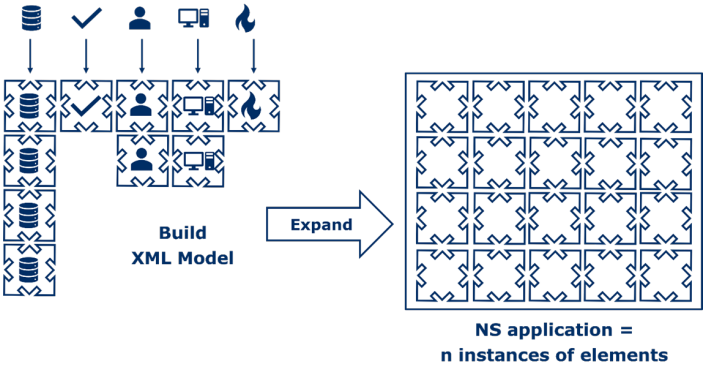

Figure 1. Requirements expressed in XML description file, used on input for element expansion.

4) Harvesting and Software Rejuvenation

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

The application can be re-expanded for different reasons. For example, the code templates of the elements are improved (e.g. fix bugs, make faster, etc.), include a new cross-cutting concern (e.g. add a new logging feature), or change the technology (e.g. 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.

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,

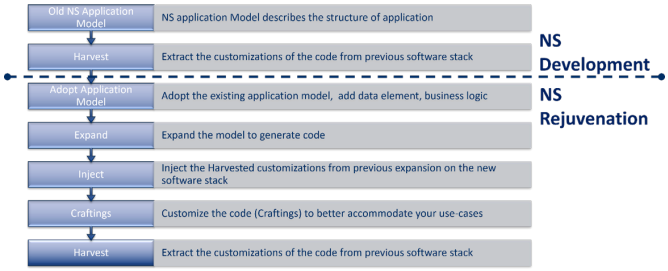

Figure 2. The NS development process.

flow, trigger and connector elements. These types of changes originate from new functionalities. The second dimension is the changes to the element code templates due to the introduction of new cross-cutting concerns or the overall improvement of the code of the templates. The third dimension is technology-induced changes, handled by the cross-cutting concerns and thus via the element templates. The fourth and last dimension represents the custom code, the crafting, which can be harvested and reinjected.

C. Clean Architecture

CA is a software design approach emphasizing code organization into independent, modular layers with distinct responsibilities. This approach aims to create a more flexible, maintainable, and testable software system by enforcing the separation of concerns and minimizing dependencies between components. CA aims to provide a solid foundation for software development, allowing developers to build applications that can adapt to changing requirements, scale effectively, and remain resilient against the introduction of bugs [6].

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

- ***Domain Layer***: This layer contains the application's core business objects, rules, and domain logic. Entities represent the fundamental concepts and relationships in the problem domain and are independent of any specific technology or framework. The domain layer focuses on encapsulating the essential complexity of the system and should be kept as pure as possible.
- ***Application Layer***: This layer contains the use cases or application-specific business rules orchestrating the interaction between entities and external systems. Use cases define the application's behavior regarding the actions users can perform and the expected outcomes. This layer coordinates the data flow between the domain layer and the presentation or infrastructure layers while remaining agnostic to the specifics of the user interface or external dependencies.

- ***Presentation Layer***: This layer translates data and interactions between the use cases and external actors, such as users or external systems. Interface adapters include controllers, view models, presenters, and data mappers, which handle user input, format data for display, and convert data between internal and external representations. The presentation layer should be as thin as possible, focusing on the mechanics of user interaction and deferring application logic to the use cases.
- ***Infrastructure Layer***: This layer contains the technical implementations of external systems and dependencies, such as databases, web services, file systems, or third party libraries. The infrastructure layer provides concrete implementations of the interfaces and abstractions defined in the other layers, allowing the core application to remain decoupled from specific technologies or frameworks. This layer is also responsible for configuration or initialization code to set up the system's runtime environment.

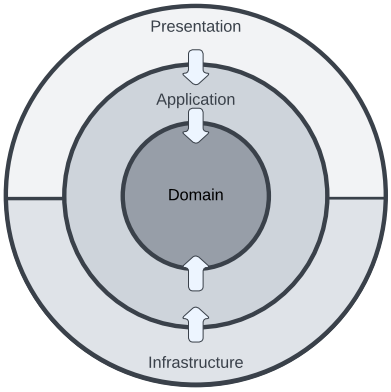

Figure 3. Flow of control

An essential aspect is described as the dependency rule. The rule states that *source code dependencies must point only inward toward higher-level policies* Martin [6, p. 206]. This 'flow of control' is designed following the Dependency Inversion Principle (DIP) and can be represented schematically as concentric circles containing all the described components. The arrows in Figure 3 clearly show that the dependencies flow from the outer layers to the inner layers. Most outer layers are historically subjected to large-scale refactorings due to technological changes and innovation. Separating the layers and adhering to the dependency rule ensures that the domain logic can evolve independently from external dependencies or certain specific technologies.

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

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

- ***Single Responsibility Principle (SRP)***: This principle has undergone several iterations of the formal definition. The final definition of the Single Responsibility Principle (SRP) is: "a module should be responsible to one, and only one, actor" Martin [6, p. 82]. The word 'actor' in this statement refers to all the users and stakeholders represented by the (functional) requirements. The modularity concept in this definition is described by Martin [6, p. 82] as a cohesive set of functions and data structures. In conclusion, this principle allows for modules with multiple tasks as long as they cohesively belong together. Martin [6, p. 81] acknowledges the slightly inappropriate name of the principle, as many interpreted it, that a module should do just one thing.
- ***Open/Closed Principle (OCP)***: Meyer [8] first mentioned the OCP and formulated the following definition: *A module should be open for extension but closed for modification.* The software architecture should be designed such that the behavior of a module can be extended without modifying existing source code. The OCP promotes the use of abstraction and polymorphism to achieve this goal. The OCP is one of the driving forces behind the software architecture of systems, making it relatively easy to apply new requirements. [6, p. 94].
- ***Liskov Substitution Principle (LSP)***: The LSP is named after Barbara Liskov, who first introduced the principle in a paper she co-authored in 1987. Barbara Liskov wrote the following statement to define subtypes Martin [6, p. 92]. *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 Martin [6, 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. 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" [1], 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" [1], 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	−	−	−	++	−

V. EXPANSION WITH CLEAN ARCHITECTURE

The primary objective of G. Koks research was to determine the degree of convergence between CA and NS Theory. To achieve this goal, the research consited out of several key objectives.

Besides the a comprehensive literature analysis, an architectural design was created, which was fully and solely based on CA principles. The findings from the literature review were

incorporated into this design, which served as the basis for the subsequent development of the research artifacts.

In the artifact development phase, two artifacts were constructed to facilitate the study of the convergence between CA and NS Theories. The first artifact was the Expander Framework and Clean Architecture Expander. These components were designed and implemented based on the CA design principles. The Clean Architecture Expander enabled the parameterized instantiation of software systems that adhere to the principles and design of CA, while the Expander Framework served as a supporting system. It was responsible for loading and orchestrating dependencies, managing models, and executing the expander.

The second artifact was the Expanded Clean Architecture artifact. This artifact allowed for the analysis of a RESTful API implementation and its alignment with CA principles and design, thereby providing a platform to evaluate the convergence of the two theories in a practical context.

Finally, the analysis of combinatorics examined the artifacts for actual or potential combinatorial effects. This analysis aimed to determine whether CA and NS exhibit convergence. The fundamental principles and architectural design of CA were considered throughout the analysis to ensure a comprehensive evaluation of the convergence potential.

By pursuing these objectives, the research provides valuable insights into the interaction between CA and NS, particularly in terms of their potential convergence within the field of software architecture.

This chapter outlines the construction of two artifacts. Both of these artifacts are meticulously designed and developed in accordance with the design philosophy and principles of CA with strict adherence to the following requirements.

A. Naming Conventions

The following section introduces the naming conventions applied throughout the project. While these conventions do not directly contribute to the stability aspects of the software architecture, they serve an important role. By adhering to consistent and descriptive naming patterns, it becomes easier to follow the structure of the code and identify key components of the artifacts. These naming conventions help readers recognize and map various elements to their corresponding roles within the CA framework, enhancing clarity and improving code comprehension without affecting the system's inherent stability.

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

TABLE IV
NAMING CONVENTION COMPONENT LAYERS

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

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

TABLE V
NAMING CONVENTION OF RECURRING ELEMENTS

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

B. Component Requirements

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

The component architecture is organized into separate Visual Studio projects for the Domain, Application, Infrastructure, and Presentation layers. A detailed description of these layers can be found in Section II-C. Each of these projects adheres to the naming conventions described in section V-A. Importantly, the dependencies between component layers must follow an inward direction, aligning with higher-level components as schematically illustrated in Figure 3. The

dependencies cannot skip layers, ensuring a clear hierarchical structure.

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

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

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

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

C. Software Architecture Requirements

Figure 4 illustrates the generic software architecture of the artifacts. Each instantiated element adheres to the Element Naming Convention outlined in section V-A. The following sections detail the requirements specific to each element.

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

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

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

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

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

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

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

Similarly, the ResponseModelMapper is responsible for mapping data attributes from the ResponseModel and follows the same implementation and scope restrictions as the RequestModelMapper.

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

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

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

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

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

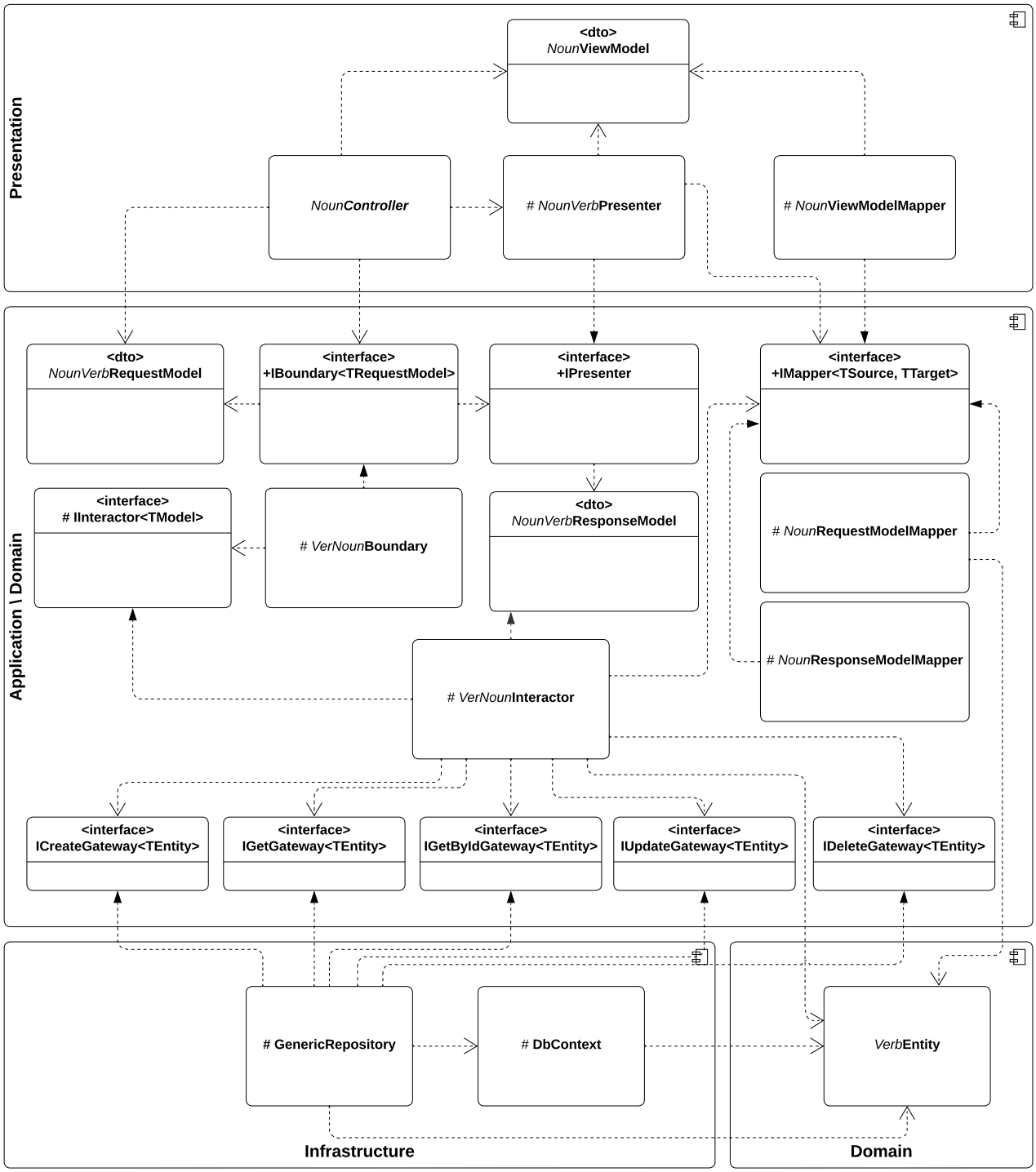

Figure 4. The Generic architecture of the artifacts

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

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

violations of these design principles.

D. Expander Requirements

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

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

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

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

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

E. Generated Artifact Requirements

The generated artifact adheres to this chapter's component and software requirements specified in Sections V-B and V-C.

VI. The Research Artifacts

The first artifact consists of two main components: the Clean Architecture Expander and the Expander framework. The name of the Expander Framework, Pantha Rhei, was inspired by the Greek philosopher *Heraclitus*, who famously stated that "life is flux." The name reflects the artifact's perceived ability to cope with constant change in a stable and evolvable manner. Users can interact with the Expander Framework using the CLI command 'flux' in combination with several parameters.

As illustrated in Figure 5, the main task of the first artifact or 'expand' the second artifact. By entering the correct command, the Expander Framework loads the model being instantiated during the expansion process. Then, the required expanders are prepared based on information available through the model. In the case of this study, the Clean Architecture Expander. The Clean Architecture Expander consists of a set of tasks and templates. When the Expander Framework executes the Clean Architecture Expander, the model is instantiated into the generated artifact with the aid of the templates.

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

As a result, the second artifact (artifact II) allows a user to modify or maintain the model used by the Expander Framework by exposing a Restful interface. This method approaches the meta-circularity process, where an expansion process is used to update the meta-model. Although not fully compliant with the theory of NS, the Expander Framework consists of the required tasks to update its own meta-model. This is illustrated in Figure 5 by the 'updates' arrow.

A. The Meta-Model and Model

The meta-model is a blueprint that describes a software system's structure, entities, relationships, and expanders. The model is an instantiation of the meta-model, representing a specific software system with unique characteristics.

Figure 6 illustrates the version of the meta-model used for this research. A detailed description of each of the elements can be found in the thesis of Koks [1, p. 73].

B. Plugin Architecture

The Expander Framework artifact is responsible for loading and bootstrapping Expanders and initiating the generation process. Expanders are dynamically loaded at runtime through a dotnet capability called assembly binding, allowing the architecture illustrated in the following image [9].

This plugin design adheres to several principles of SOLID. The SRP principle is implemented by ensuring that an expander generates one and only one construct. The OCP principle is applied by allowing the creation of new expanders in addition to the already existing ones. The LSP principle is respected by enabling the addition or replacement of expanders without modifying the internal workings of the Expander Framework.

C. Expanders

The Exander Framework allows for the miscellaneous execution of expanders of any type. The Expander Framework is independent of any of the details of Expanders, fully adhering to the principle of DIP. Conversely, an Expander is required to implement several interfaces to ensure execution and dependency management are available during runtime. The Expander Framework also consists of a set of default tasks, such as the execution of the expansion tasks known as ExpanderHandlerInteractors *IExpanderHandlerInteractor* [10], logging, bootstrapping dependencies, and tasks to execute harvestings and injections. Except for the use of the IExpanderInteractor, non of which are required.

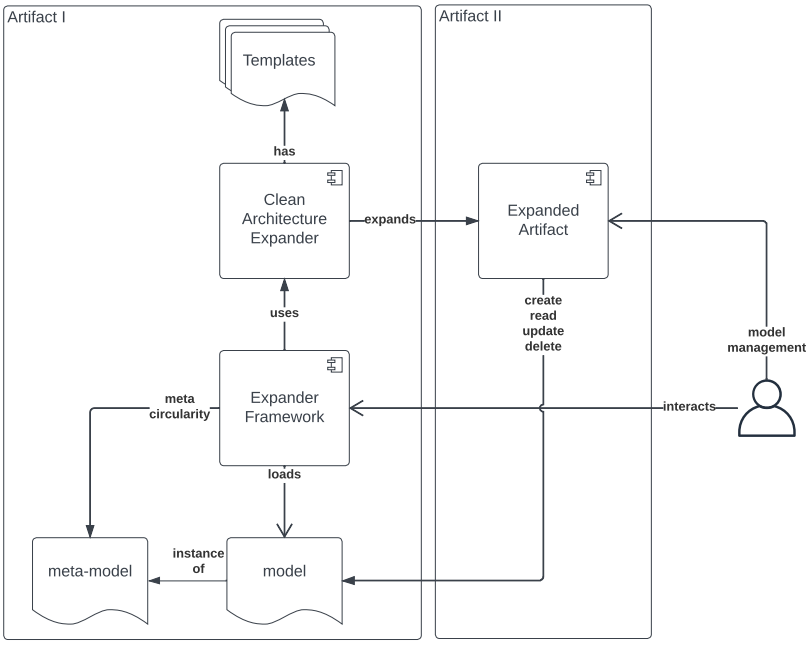

Figure 5. Schematic overview of the artifacts

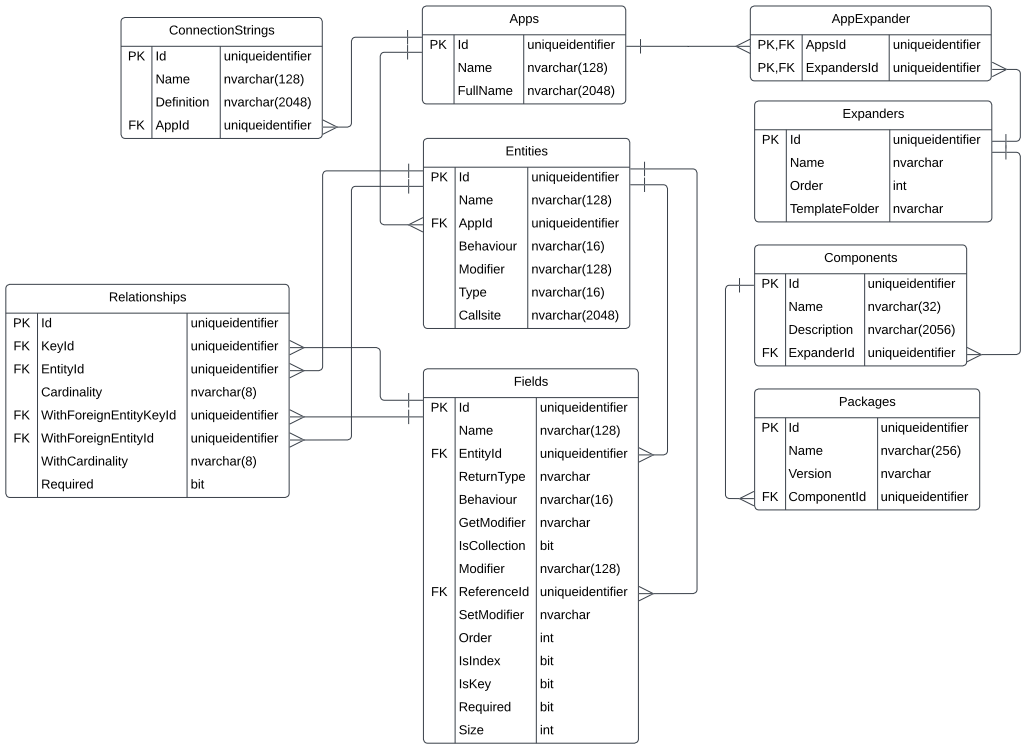

Figure 6. The meta-model represented as an Entity Relationship Diagram

Figure 8 illustrates the dependencies between the domain layer of the Expander Framework. The Clean Architecture Expander is considered an application layer containing specific tasks bounded to a particular application or process. In this case, the Expansion process.

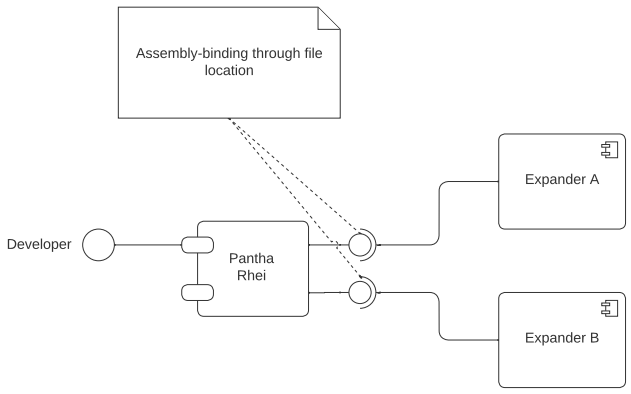

Figure 7. Expanders are considered plugins

D. Executing Commands

An implementation that facilitates a high degree of cohesion while maintaining low coupling is the utilization of the IExecutionInteractor interface [11]. This interface allows for the execution of various derived types responsible for various tasks, such as executing Handlers, Harvesters, and Rejuvenators[1] [12]–[14]. The implementation promotes decoupling by adhering to both OCP and LSP.

Figure 9 illustrates that the required interfaces are placed in the Domain layer of the Expander Framework. In contrast, the concrete classes also can be implemented as part of the internal scope of the Clean Architecture Expander [15]. The artifact illustrates the aggregation of the execution, which allows for a graceful cohesion of the execution Tasks [16].

E. Dependency Management

Dependency management is an extremely valuable aspect of achieving stability and evolvability. Dependency management can be achieved by using Dependency Injection. This research acknowledges Mannaert *et al.* [7, p. 215] statement that Dependency Injection does not solve coupling between classes. Working on the artifact has shown that combinatorial effects can occur when not careful. Nevertheless, Dependency Injection is a widely used pattern in building the artifact. In order to achieve stability and evolvability, the Dependency Injection pattern must be combined with various other principles of both CA and NS.

The goal is to centralize the management of dependencies and remove unwanted manual object instantiations in the code. Al this while respecting the DIP principle so that each component layer is responsible for managing its dependencies. The artifact achieves this by using extension methods [17]. Additionally, and quite significantly, implementations primarily rely on abstractions or contracts (interfaces) instead of the details of concrete implementations.

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

Traditionally, Dependency Injection injects instantiations through constructor parameters or class properties. Although there are benefits in this approach, doing so will eventually lead to combinatorial effects, breaking the stability of a software artifact. In order to solve this problem, the artifact used the Service Locator pattern, a central registry responsible for resolving dependencies [0]. Many frameworks are available from Nuget.org, but the artifact uses the Service Registry, which is part of the .NET framework. This service registry is considered a cross-cutting concern. The dependency on this technology is reduced by applying the principles of the LSP and ISP. The artifact creates and uses separate interfaces to register [18] and resolve [19] dependencies. The framework technology dependencies are abstracted behind implementing those interfaces [20].

The approach described here has many advantages in managing the stability and evolvability of the software artifact. However, as for most things, there are also some drawbacks. For example, a good amount of experience is required for developers to understand code that incorporates abstractions, contracts, and Dependency Injection. Another drawback is that dependency errors are detected in runtime rather than compile time. The benefits of the artifacts, however, outweigh the drawbacks.

VII. 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.

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. Throughout this paper, NS is referred to as a development approach or paradigm, 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 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.

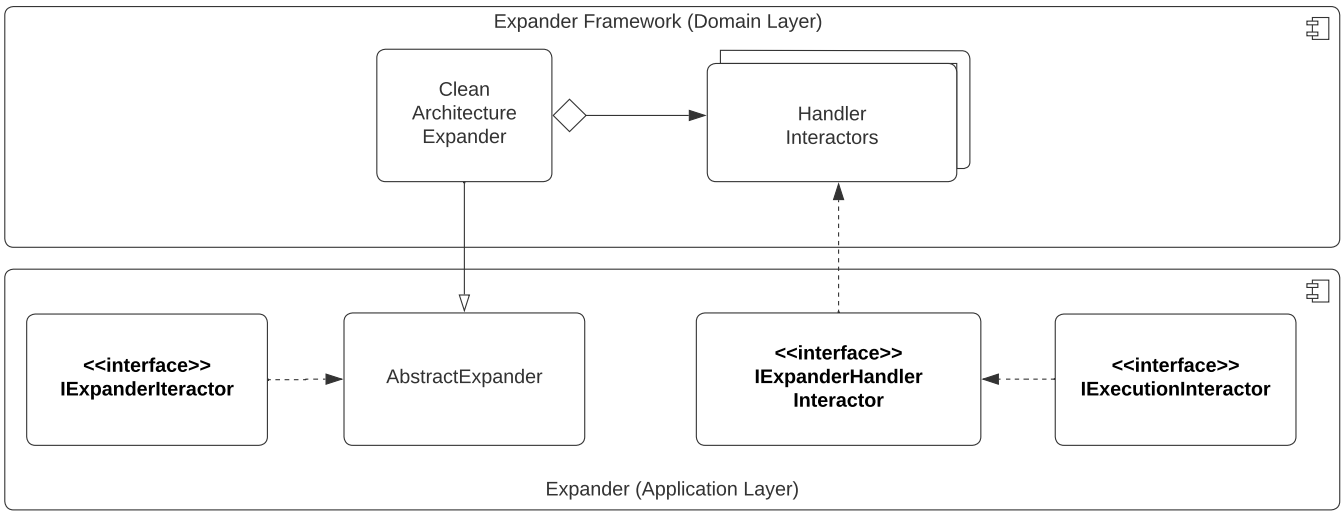

Figure 8. The design of an Expander

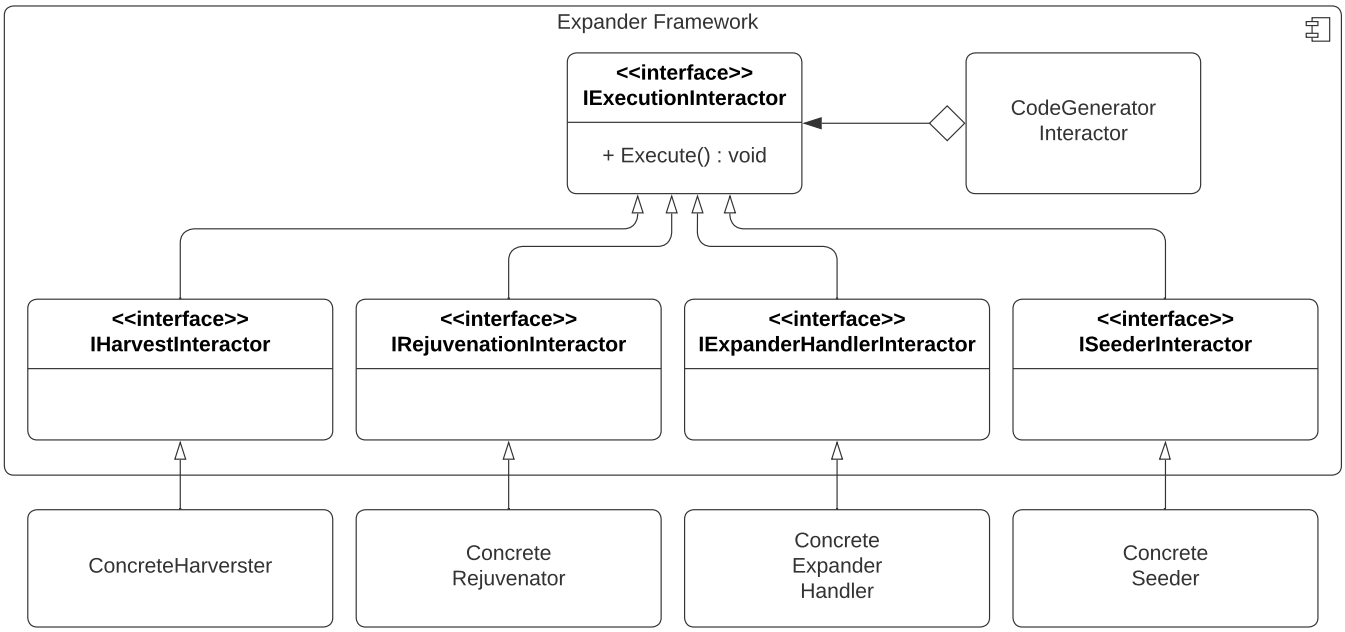

Figure 9. Low coupling with IExecutionInteractor

The underrepresentation of DvT has led to significant combinatorial effects in some parts of the researcher's artifacts. These combinatorial effects might be attributed to the author's inexperience in creating systems that enable code generation through expansion while maintaining stability on templates and craftings. If DvT were better represented in the principles of CA, the severity of the combinatorial effects would have most likely been less.

CA Lacks a strong foundation for receiving external triggers in its design philosophy. This is partially represented by the Controller element. However, this element is described as being used for web-enabled environments and might result in a less comprehensive approach to receiving external triggers across various technologies or systems.

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

dependencies and side effects.

The findings can only lead to the conclusion that the convergence between CA and NS is incomplete. Consequently, CA cannot fully ensure stable and evolvable software artifacts as NS has defined them.

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

CA offers a well defined, practical, and modular structure for software development. Its principles, such as SOLID, guide developers in creating maintainable, testable, and scalable systems. This architectural design approach is highly suitable for various applications and can be easily integrated with the theoretical foundations provided by NS. Conversely, the NS approach offers a more comprehensive theoretical understanding of achieving stable and evolvable systems.

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

Bibliography

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

[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. Accessed: 2023-03-19.

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

[7] H. Mannaert, J. Verelst, and P. De Bruyn, *Normalized systems theory: from foundations for evolvable software toward a general theory for evolvable design*, eng. Kermt: nsi-Press powered bei Koppa, 2016, ISBN: 978-90-77160-09-1.

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

Code Samples

[9] G. Koks, *ExpanderPluginLoaderInteractor*, 2023. Accessed: 2023-05-01. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Application/Interactors/Initializers/ExpanderPluginLoaderInteractor.cs.

[10] G. Koks, *IExpanderHandlerInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/IExpanderHandlerInteractor.cs.

[11] G. Koks, *IExecutionInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/IExecutionInteractor.cs.

[12] G. Koks, *ExpandEntitiesHandlerInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Expanders/src/PanthaRhei.Expanders.CleanArchitecture/Handlers/Domain/ExpandEntitiesHandlerInteractor.cs.

[13] G. Koks, *RegionHarvesterInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Harvesters/RegionHarvesterInteractor.cs.

[14] G. Koks, *RegionRejuvenatorInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Rejuvenator/RegionRejuvenatorInteractor.cs.

[15] G. Koks, *MigrationHarvesterInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Expanders/src/PanthaRhei.Expanders.CleanArchitecture/Harvesters/MigrationHarvesterInteractor.cs.

[16] G. Koks, *CodeGeneratorInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Application/Interactors/Generators/CodeGeneratorInteractor.cs.

[17] G. Koks, *DependencyInjectionExtension*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Application/DependencyInjectionExtension.cs.

[18] G. Koks, *IDependencyManagerInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain/Interactors/Dependencies/IDependencyManagerInteractor.cs.

[19] G. Koks, *IDependencyFactoryInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain/Interactors/Dependencies/IDependencyFactoryInteractor.cs.

[20] G. Koks, *DependencyManagerInteractor*, 2023. [Online]. Available: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain/Interactors/Dependencies/DependencyManagerInteractor.cs.