ANTWERP MANAGEMENT SCHOOL

On the convergence of Clean Architecture with the Normalized Systems Theorems

A Design Science approach of modularity, stability and evolvability of a C# software artifact.

Author:
Gerco Koks

Supervisor:
Prof. Dr. Ing. Hans Mulder

Promotor:
Dr. ir. Geert Haerens

Co-Promotor:
Frans Verstreken, MSc

A thesis submitted in fulfillment of the requirements for the degree of Master of Enterprise IT Architecture

April 16, 2023

Declaration of Authorship

I, Gerco Koks, declare that this thesis titled, “On the convergence of Clean Architecture with the Normalized Systems Theorems” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for the quotations, this thesis is entirely my work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

ANTWERP MANAGEMENT SCHOOL

Abstract

Master of Enterprise IT Architecture

On the convergence of Clean Architecture with the Normalized Systems Theorems

by Gerco Koks

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

For/Dedicated to/To my…

1 Introduction

Sinds the early days there have been challenges with creating and maintaining stable and evolvable software. On the one hand, this is caused by constantly evolving requirements as new business opportunities, technologies, methodologies, and best practices are developed to meet the demands of modern corporate environments. On the other hand, changing software can lead to deterioration in stability and evolvability, which can negatively impact the quality of these systems.

These challenges have been recognized by early pioneers in software engineering. The laws of software evolution describe the balance between the forces driving new requirements and those that slow down progress (Lehman 1980).

Back in 1969, Douglas McIlroy proposed a vision where the systematic reuse of software building blocks should lead to more reuse, a very early concept of modular software constructs (P. Naur and B. Randell 1968, p. 79). He quoted "The real hero of programming is the one who writes negative code" (i.e., when a change in a program source makes the number of lines of code decrease ('negative' code), while its overall quality, readability or speed improves) [1].

Dijkstra argued against using unstructured control flow in programming and advocated for using structured programming constructs to improve the clarity and maintainability of the source code (Dijkstra 1968). He advocated structured programming techniques that improved the modularity and evolvability of software artifacts. Parnas continued with the principle of information hiding. He stated that design decisions used multiple times by a software artifact should be modularized to reduce complexity (Parnas 1972).

Various programming paradigms, including procedural, object-oriented, and functional programming, have emerged to enhance software programming capabilities that contribute to stability and evolvability. In addition, these paradigms have impacted modern programming languages, such as Java and C#, enabling the development of more modular and evolvable software architectures.

Design principles, patterns, and theorems are, on top of all, additional measures to enhance the modularity, stability, and evolvability of software artifacts. As a junior software engineer, I was always intrigued by the concepts of quality and maintainable code and quickly got introduced to the Single Responsibility Principle, Open/-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle (SOLID) principles. And later on with the complete design approach derived from Clean Architecture. Starting my Master's degree, I got an inspiring introduction from Jan Verelst and realized quality and maintainability were essentially all about the concepts of stability and evolvability. For me, it was very interesting that the Normalized Systems Theory is supported by empirical scientific evidence. Although I'm not that active anymore in the field of software

[1] `https://en.wikipedia.org/wiki/Douglas_McIlroy`

engineering, I immediately knew the topic of my research. it is still a big passion of mine.

Given my experience with Clean Architecture, and what I was learning from Normalized Systems Theory, I noticed a lot of similarities, but also some big differences. In early investigations, I found overlapping characteristics. But it seemed there were also a couple of differences. I wanted to know if the design approaches could be used in conjunction with each other, perhaps bettering the result of stable and evolvable software.

Java SE has primarily been used for case studies in order to develop the Normalized Systems Theory (Oorts et al. 2014; De Bruyn et al. 2018). Although sufficient in Java, I was pleased to read that both software design approaches have formulated their modular structures independent of any programming technology (Mannaert and Verelst 2009; Robert C. Martin 2018). So I was free to use my favorite programming language C# to create a software artifact that supported my research, igniting my passion for programming again.

Based on my early investigations of both design approaches I hypothesized that they can be used in conjunction with each other. Consequently, an artifact that is designed based on the principles of Clean Architecture will lead to a highly modular, stable, and evolvable C# artifact that does not contradict a design based on the Normalized Systems theorems.

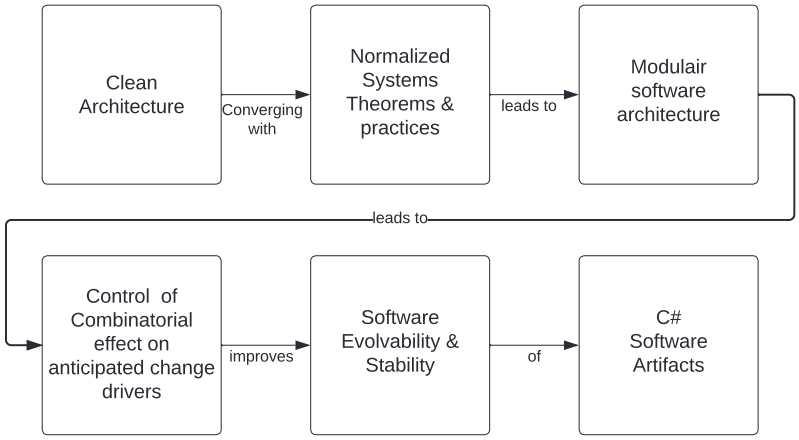


FIGURE 1.1: The hypothesis

1.1 Research Objectives

In this Design Science Research, we will shift the focus from Research Questions to Research Objects. The Object of this research is to determine the degree of convergence of Clean Architecture, with the Normalized Systems Theory.

To summarize, we will start with a generic design that is based on the principles and design characteristics of Clean Architecture. Given this design, we will conduct the research on two different artifacts, namely a code-generator artifact and a generated artifact. The reason for the code-generator artifact is to provide strict and meticulous adherence to the generic design. Each entity should be implemented in exactly the same way.

AFMAKEN

1.2 Research method

This research is a Design Science Method and relies on the Engineering Cycles as described by Wieringa. The engineering cycle provides a structured approach to develop the required artifacts to analyze the design problem (Wieringa 2014).

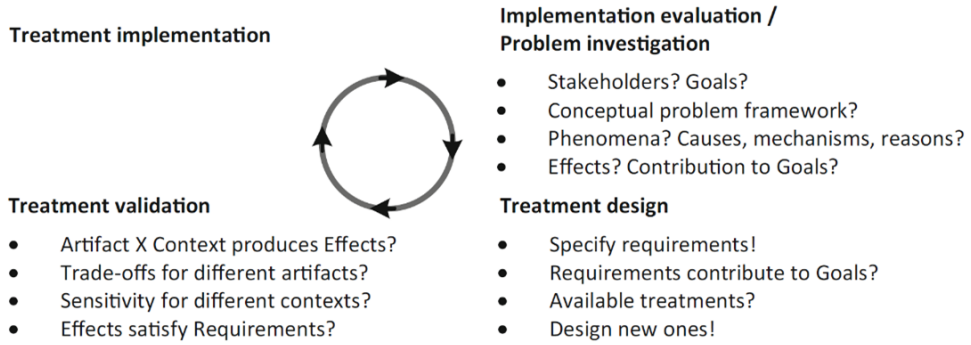


FIGURE 1.2: Wieringa's engineering cycle

In the context of this research, the artifacts described in chapters 4.1 and 4.2 are considered to be information systems. Hevner et al. proposed a framework for research in information systems by introducing the interacting relevance and rigor cycles.

Figure 1.3 depicts a specialized overview of Hevners Design Science Framework. The rigor cycle is composed of the theories and knowledge from NS and CA. This is supplemented by the rigorous knowledge of modularity, evolvability, and stability of software systems. The Relevance cycle represents the business needs of the stakeholders. The business needs are described as research objectives, research questions, and research requirements.

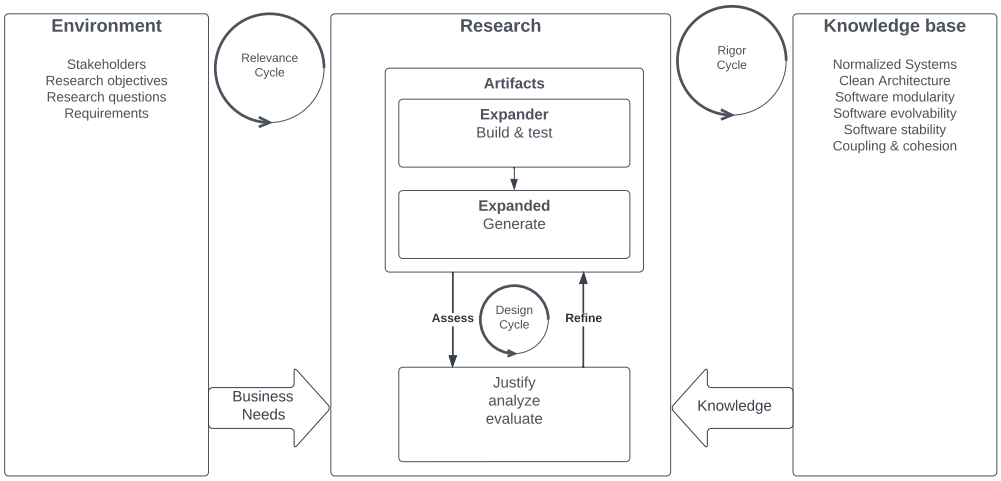


FIGURE 1.3: The Design Science Framework for IS Research

1.3 Thesis outline

The structure of this thesis reflects the research methodology described in the previous section 1.2. Chapter 2 presents the theoretical backgrounds of both NS and CA, discussing important characteristics and requirements of software stability, as well as the principles and architectures proposed by both development approaches.

Chapter 3 focuses on the requirements relevant to this research. It is divided into two sections: section 3.1 outlines the research requirements, describing the requirements necessary for conducting the research, while section 3.2 details the artifact requirements, laying out the requirements relevant to the artifacts contributing to this research.

Chapter 5 evaluates the research results, discussing the impact of using CA on NS. Notable experiences and findings in Chapter 6 Discussion. The conclusion of this research is presented in the final chapter, Chapter 7.

Lastly, it is worth mentioning that this thesis follows the guidelines of the American Psychological Association (APA) style, including the use of US spelling.

2 Theoretical background

The goal of this thesis is to investigate whether CA approach aligns with the goals of NS. Therefore, it is essential to have a comprehensive understanding of software stability and the key concepts, principles, and architectures that impact software stability.

This chapter begins by examining the concepts of software stability, evolvability, and modularity, highlighting their significance in achieving software stability in NS. This is followed by a brief overview of the design theorems and proposed architecture of NS.

The subsequent sections of the thesis explore the fundamental principles that underlie CA, as well as its proposed architectural designs. Finally, the thesis concludes by discussing which aspects of CA align with the principles of NS and contribute to achieving software stability in this approach.

2.1 Normalized Systems: Impacting software stability

NS is a software development approach that prioritizes achieving software stability through the use of standardized, modular components and interfaces. This theory is informed by several scientific disciplines, including systems theory, mathematics, and computer science, as well as some other software development approaches, such as agile development and domain-driven design.

NS originated in the field of software engineering. However, the underlying theory of NS can be applied to various other domains, such as Enterprise Engineering, Business Process Modeling, and document management. This research acknowledges the software engineering background of NS. It consistently refers to software and Information Systems when referring to 'artifacts.' However, the reader should realize that the concepts and artifacts are not restricted to software artifacts alone.

2.1.1 Towards stability

In several disciplines, stability has been defined as *Bounded Input Bounded Output* (BIBO). It is the fundamental property of a system when subjected to bounded input disturbances. BIBO stability ensures that the output of a system will also be bounded, preventing uncontrolled or unexpected behavior (Mannaert et al. 2016, p. 270).

A real-world example of the importance of stability is the Tacoma Narrows Bridge in Washington State, USA. This bridge, depicted in Figure 2.1, collapsed on November the 7th, 1940. This was caused due to wind-induced oscillations called aeroelastic

flutter. The wind (Input) induced oscillations in the bridge, causing it to start swaying back and forth (Output). These oscillations were initially small, but as they continued, they began to increase in amplitude and magnitude. Eventually, this caused the bridge to collapse.

FIGURE 2.1: Tacoma Narrows Bridge (Galloping Gertie)

Stability can also be used in the context of software engineering. In the context of NS, it is considered a critical property that ensures that the software is not excessively sensitive to small changes (Mannaert et al. 2016, p. 270). New functional requirements should only lead to a fixed and expected amount of changes in the source code.

Conversely, instabilities occur when the total number of modifications relies on the size of the software artifact. When there are software instabilities, the number of changes will grow over time in parallel with the growth of the system. These instabilities are referred to as combinatorial effects (Mannaert et al. 2016, p. 270).

When combinatorial effects are absent, the software artifact can be considered evolvable.

2.1.2 Towards evolvability

In NS, evolvability is a crucial property to achieve stable software systems. An evolvable system can adapt over time in response to changing requirements. NS attempts to achieve evolvability by providing guidelines, principles, and theorems n order to achieve a modular (and scalable) architecture that allows for easy adaptability, extensibility, and the replacement of components with a minimum impact on the quality of the functionality and the overall structures of the architecture. This is achieved through the use of formalized models that define the system's components, interfaces, and behavior, as well as through the separation of concerns between different parts of the system.

There are several aspects concerning the evolvability of software systems. One of which is the modularity of the architecture. There is also a broad consensus about two fundamental rules when thinking of- and designing modularity: *high cohesion* and *low coupling* (Mannaert et al. 2016, p. 22).

2.1.3 Modularity

A Software module can be defined as self-contained units of code that perform specific tasks or sets of tasks within a more extensive system. A software module is designed to operate independently of other modules, with well-defined interfaces that allow it to communicate and exchange data with other modules if necessary (Mannaert et al. 2016, p. 22).

A module can be considered a hierarchical and recursive concept. They are independent of their size (lines of code) or computational magnitude. They can be as small as a function as part of a class. The class itself can also be considered a module. A group of classes contained in a Dynamic Link Library (DLL) or Application Programming Interface (API) can also be considered a module of an even more extensive system.

An essential part of the design of a software system is to identify the possible different modules and their interaction interfaces. Figure 2.4 presents a high-level depiction of modular manifestations in the artifact, with additional examples of modularity found in more granular implementations. Further discussion of this architecture is provided in Chapter 2.2.

2.1.4 Cohesion

The term cohesion denotes the extent to which the various structural components of a software system operate cohesively towards a singular and well-defined objective or goal. Empirical studies in software engineering have extensively demonstrated the significance of cohesion, linking higher levels of cohesion with reduced defects, enhanced maintainability, and greater openness to change. Consequently, achieving high cohesion has been associated with an overall improvement in software quality attributes such as reliability, maintainability, reusability, and evolvability.

Cohesion facilitates the reduction of complexity and interdependence among the components of a system, thereby contributing to a more efficient, maintainable, and reliable system. By organizing components around a shared purpose or function or by standardizing their interfaces, data structures, and protocols, cohesion can offer the following benefits:

- **Reduce redundancy and duplication of effort**:
  Cohesion ensures that components are arranged around a common purpose or function, reducing duplicates or redundant code. This simplifies system comprehension, maintenance, and modification.
- **Promoting code reuse:**
  Cohesion facilitates code reuse by making it easier to extract and reuse components designed for specific functions. This saves time and effort during development and enhances overall system quality.
- **Enhance maintainability:**
  Cohesion decreases the complexity and interdependence of system components, making it easier to identify and rectify bugs or errors in the code. This improves system maintainability and reduces the risk of introducing new errors during maintenance.

- **Increase scalability:**
  Cohesion improves a system's scalability by enabling it to be extended or modified effortlessly to accommodate changing requirements or conditions. By designing well-organized and well-defined components, developers can easily add or modify functionality as needed without disrupting the rest of the system.

2.1.5 Coupling

Coupling is an essential concept in software engineering that pertains to the degree of interdependence among software modules and components. The level of coupling between modules denotes the strength of their relationship, whereby a high level of coupling implies a significant degree of interdependence. Conversely, low coupling signifies a weaker relationship between modules, where modifications in one module are less likely to impact others. Although not always possible, the level of coupling between the various modules of the system should be kept to a bare minimum.

The negative impact of excessive coupling on software systems is considerable. High coupling can render software systems challenging to maintain, modify, or evolve. It can make it considerably more challenging to find the root cause of potential bugs. Additionally, it causes fragility in the system, where slight modifications in one module can trigger cascading failures throughout the entire system. Therefore, it is crucial for software engineers to minimize coupling between modules while maintaining a cohesive design. By developing systems with low coupling, software engineers can construct more maintainable, scalable, and adaptable systems that are easier to evolve.

Coupling in software engineering can take several forms, including content, common, control, stamp, and data coupling. Content coupling occurs when one module accesses or modifies the internal data or logic of another module, leading to high interdependence and difficulty in isolating errors. Common coupling occurs when several modules access and use the same global data, increasing their interdependence and reducing modularity. Control coupling occurs when one module controls the execution flow of another module, making it challenging to modify or reuse the controlled module. Stamp coupling arises when two modules share a common data structure, leading to tight coupling and high interdependence. Finally, data coupling exists when two modules share data, which can lead to coupling between them.

One attempt to lower coupling in the expanded artifact is to prefer stamp coupling over data coupling through the API interface. This is done by making use of RequestModels and ViewModels, instead of the actual data element (see example in Listings 2.1). Depending on the use case, only the required data is passed down to the view, or in the case of a command accepted as an input parameter.

```
1  /// <summary>
2  /// The actual data entity of the component.
3  /// </summary>
4  public class Component
5  {
6      public virtual Guid Id { get; set; }
7      public virtual string Name { get; set; }
8      public virtual string Description { get; set; }
9      public virtual List<Package> Packages { get; set; }
10     public virtual Expander Expander { get; set; }
11 }
12
```

```
/// <summary>
/// The Component entity represented as a ViewModel.
/// </summary>
public class ComponentViewModel : ViewModel
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public List<PackageViewModel> Packages { get; set; }
    public ExpanderViewModel Expander { get; set; }

    #region ns-custom-properties
    #endregion ns-custom-properties
}

/// <summary>
/// The RequestModel for the Delete command for a Component entity.
/// </summary>
public class DeleteComponentCommand : RequestModel
{
    public Guid Id { get; set; }
}
```

LISTING 2.1: The ViewModel (Koks 2023f) and RequestModel (Koks 2023g) of the Entity 'Component' (Koks 2023w)

2.1.6 Expansion and code generation

Creating and maintaining a stable and evolvable system is a particularly challenging and meticulous engineering job. Developers are required to have a sound knowledge of NS, whilst implementing new requirements in an always consistent manner. Given the recurring structures, processing new requirements is a very precise, strict and meticulous job. (Mannaert et al. 2016, p. 403) Manual labor could be error-prone given modern time-to-market requirements.

Therefore, it seems logical to automate the instantiation process of software structures and use code generation for recurring tasks (Mannaert et al. 2016, p. 403). This is where the concepts of code generation and expansion come into place. This does not only refer to the automatic process of adapting and maintaining software to new requirements, architectural enablers and technological alterations. It also embraces manually added craftings to the software, the so-called plugin code. These craftings are preserved after each expansion by a method that is called harvesting and rejuvenation (Mannaert et al. 2016, pp. 405–406).

2.1.7 The Theoretical Framework

NS consists of a theoretical framework describing a set of design principles. These principles are the basis for achieving the concepts of stability, evolvability, and modularity. NS provides a rigorous and mathematical foundation for these theorems and they offer guidelines for designing and developing software systems. In the following sections, we will focus on the principles of NS very briefly as they have been extensively described in various scientific papers.

We know from Chapter 3.2 that the design artifacts, as a part of this research, are implemented based on the CA principles. Therefore, contrary to Chapter 2.2, there will be no references to the manifestations of the NS design theorems in the design.

Separation of Concerns

Separation Of Concerns (SOC) as a principle has first been mentioned by Dijkstra[1] as the crucial principle to design modular software architecture (Dijkstra 1982). SOC promotes the idea that a program should be divided into distinct sections, each addressing a particular concern or aspect of a design problem. This allows for a more organized and maintainable source code. When implemented correctly, a change to one concern does not affect the others. SOC should be applied at the level of individual modules, rather than the level of an entire program.

SOC has been adopted as one of the design theorems of NS, although it has a stricter definition of this principle(Mannaert et al. 2016).

Theorem I

A processing function can only contain a single task to achieve stability.

Data version Transparency

Data Version Transparency (DvT) is the act of encapsulation of data entities for specific tasks at hand. This results in the fact that data structures can have multiple versions often mentioned as Data Transfer Objects in modern software engineering projects. In other words, it should be possible to update the data entity without affecting the processing functions. This leads to the following description of the theorem (Mannaert et al. 2016, p. 280).

Theorem II

A data structure that is passed through the interface of a processing function needs to exhibit version transparency to achieve stability.

DvT is widely used in various technological applications. practically every web service currently known supports some type of versioning. In restful APIs, for example, it is common practice to support versioning over the URI. It is considered a best practice to encapsulate breaking changes in a new version of the endpoint/service so that the consumers are not (directly) affected by the change. In modern Object Oriented languages, glsdtv is also supported by the ability to determine the scope of visibility of the modifiers of the various programming constructs like fields, properties, interfaces, and classes, also known as information hiding (Parnas 1972; Mannaert et al. 2016, p. 278).

Action version Transparency

Action Version Transparency (AvT) is the property of a system to modify existing processing functions without affecting the existing ones. It should be possible to upgrade a function without affecting the callers of those functions. This description leads to the following theorem (Mannaert et al. 2016, p. 282).

Theorem III

A processing function that is called by another processing function, needs to exhibit version transparency to achieve stability.

[1] `https://en.wikipedia.org/wiki/Separation_of_concerns`

Most of the modern technology environments support some form of AvT. Polymorphism is a widely used technique to support this theorem. Specifically, parametric polymorphism [2] allows for a processing function to have multiple input parameters. There are also quite some design patterns supporting this theorem. Some random examples are the state pattern [3], facade pattern [4] and observer pattern [5].

Separation of State

Separation of State (SOS) is a theorem that is based on the idea that processing functions should not contain any state information but instead should rely on external data structures to store state information. By separating state information from processing functions, Normalized Systems can achieve a higher level of flexibility and adaptability. External data structures can be updated or replaced without affecting the processing functions themselves, which significantly reduces the change of unwanted ripple effects. This theorem is described as followed: (Mannaert et al. 2016, p. 258).

Theorem IV

Calling a processing function within another processing function, needs to exhibit state keeping to achieve stability.

2.1.8 Normalized Elements

In the context of the NS Theory approach, the goal is to design evolvable software, independent of the underlying technology. Nevertheless, when implementing the software and its components, a particular technology must be chosen. For Object Oriented Programming Languages like Java, the following Normalized Elements have been proposed (Mannaert et al. 2016)[363-398].

This research's artifacts utilized C# as the primary programming language. It is essential to recognize that different programming languages may necessitate alternative constructs (Mannaert et al. 2016)[364]. Given the strong similarities between C# .NET and Java, it is assumed that the same Normalized Elements are applicable for C# .NET implementation of this research's artifacts.

The Data Element

This is an object that represents a piece of data in the system. Data elements are used to pass information between processing functions and other objects. In NS, data elements are typically standardized to ensure consistency across the system.

The Task Element

This is an object that represents a specific task or action in the system. Tasks can be composed of one or more processing functions and can be used to represent complex operations within the system.

[2] https://en.wikipedia.org/wiki/Parametric_polymorphism
[3] https://en.wikipedia.org/wiki/State_pattern
[4] https://en.wikipedia.org/wiki/Facade_pattern
[5] https://en.wikipedia.org/wiki/Observer_pattern

The Connector Element

This object is used to connect different parts of the system. Connectors can be used to link processing functions, data elements, and other objects, allowing them to work together seamlessly.

The Flow Element

This object represents the flow of control through the system. It determines the order in which processing functions are executed and can be used to handle error conditions or other exceptional cases.

The Trigger Element

a trigger element is an object that reacts to specific events or changes in the system by executing predefined actions.

2.2 Clean architecture: A design approach

CA is a software design approach that emphasizes the organization of code into independent, modular layers with distinct responsibilities. This approach aims to create more flexible, maintainable, and testable software systems by enforcing the separation of concerns and minimizing dependencies between components. The goal of clean architecture is to provide a solid foundation for software development, allowing developers to build applications that can adapt to changing requirements, scale effectively, and remain resilient against the introduction of bugs (Robert C. Martin 2018).

2.2.1 The Design principles

Robert C. Martin argues that, without a solid (pun intended) set of design principles a design and architecture can quickly turn into a well-intended mess of bricks and building blocks. This is where the SOLID design principles come into place.

SOLID is an acronym for SOLID. The SOLID principles guide the developer on how to arrange the architecture of the software system. It can be considered a set set of rules on how to arrange data structures and functions into classes (Robert C. Martin 2018, p. 78).

The upcoming sections will provide a brief overview of each of the SOLID principles, as there is a plethora of literature on this subject. In chapter 3.2 we learn that one of the requirements is to design the artifact solely based on the design approach of CA. As such, each principle's description will be accompanied by one or more manifestation examples in the artifact, aligning with the research objectives.

The Single Responsibility Principle

The SRP is one of the fundamental design principles of CA. The principle advocates designing systems with high cohesion and low coupling. The SRP states that each module in a system should have only one reason to change. That is, it should have a single responsibility. No matter the granularity of the module, so implementations of methods, classes, libraries and architecture layers should adhere to SRP. By adhering to the principle, each module becomes highly cohesive, meaning that

its responsibilities are closely related and well-defined, while also being decoupled from other modules (Robert C. Martin 2018, p. 81).

The final statement of SRP is as followed (Robert C. Martin 2018, p. 82).

Single Responsibility Principle

A module should be responsible to one, and only one, actor.

SRP is closely related to the concept of SOC, which also advocates separating a system into distinct parts. Although not that clearly stated in the literature, Robert C. Martin argues that SOC intends to have a separation on a functional level and architectural level. This divides a system into different layers or components based on their functional roles. SRP is concerned with separating the responsibilities of individual modules regardless of the granularity of the module. (Robert C. Martin 2018, p. 205).With this in mind, SRP adheres more to the definition of SOC from NS. *A processing function can only contain a single task to achieve stability.* (see chapter 2.1.7 Separation of Concerns).

There are various manifestations of SRP implemented in the artifacts. One of which is already mentioned in Figure 2.4, where SRP is applied to separate the domain logic from the application, infrastructure and presentation logic. One could argue that this manifestation is more related to SOC, considering the high granularity of the components.

A better example is the separation of handlers that are part of the CA Expander. Each of those handlers executes an isolated part of the expanding process. Consider the Listing 2.2 The *ExpandEntitiesHandlerInteractor* (Koks 2023h) for example. This Handler is solely responsible for the generation of data entities.

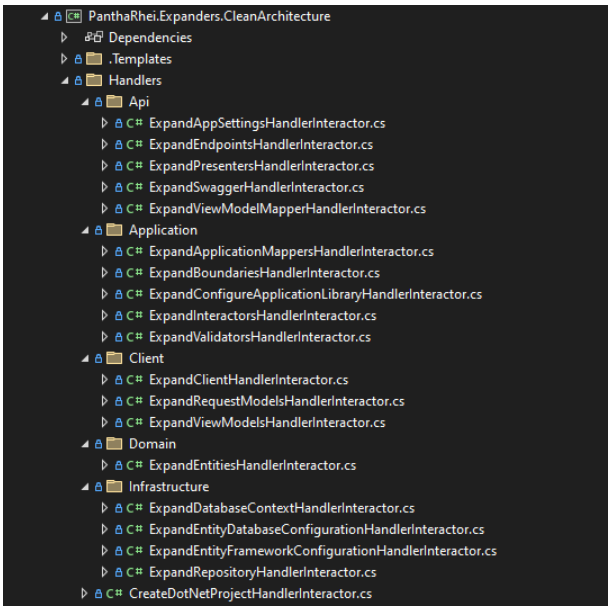


FIGURE 2.2: Each of the handlers handles an isolated part of the expanding process.

```
public class ExpandEntitiesHandlerInteractor
    : IExpanderHandlerInteractor<CleanArchitectureExpander>
{
    // ... other code
```

```
5
6       /// <inheritdoc/>
7       public void Execute()
8       {
9           directory.Create(entitiesFolder);
10
11          foreach (var entity in app.Entities)
12          {
13              string fullSavePath = Path.Combine(
14                  entitiesFolder,
15                  $"{entity.Name}.cs"
16              );
17
18              templateService.RenderAndSave(
19                  pathToTemplate,
20                  new { entity },
21                  fullSavePath
22              );
23          }
24      }
25  }
```

LISTING 2.2: The *ExpandEntitiesHandlerInteractor*

The Open-Closed Principle

The OCP was first mentioned by Meyer. He described the principle as followed (Meyer 1997, p. 79). The reference here is for the second edition of the book, the original version is from 1988.

Open/Closed Principle

A module should be open for extension but closed for modification.

OCP emphasizes the importance of designing systems that are open for extension but closed for modification. This means that the behavior of implementations can be extended without modifying its source code. The OCP promotes the use of abstraction and polymorphism to achieve this goal. By using interfaces, and abstract classes a system can be designed to allow for new behaviors to be added through extension, without changing the existing code. The OCP is one of the driving forces behind the architecture of systems. The goal is to make the system easy to extend without incurring a high impact of change (Robert C. Martin 2018, p. 94).

A relevant manifestation of OCP are all the different implementations of expander handlers in figure 2.2. The availability of the IExpanderHandlerInteractor interface makes it possible to add more functionality to the CleanArchtictureExpander without modifying any existing implementation. New handlers are added by extension, and when implemented correctly, the handler is automatically executed in the desired order and the required conditions.

```
1  /// <summary>
2  /// Specifies the interface of an expander handler.
3  /// </summary>
4  /// <typeparam name="TExpander"><seealso cref="IExpanderInteractor"/></
       typeparam>
5  public interface IExpanderHandlerInteractor<out TExpander> :
       IExecutionInteractor
6      where TExpander : class, IExpanderInteractor
7  {
8      /// <summary>
9      /// Gets the name of the <see cref="IExpanderHandlerInteractor{TExpander
           }"/>.
10     /// </summary>
```

```
11      string Name { get; }
12
13      /// <summary>
14      /// Gets the order in which the handler should be executed.
15      /// </summary>
16      int Order { get; }
17
18      /// <summary>
19      /// Gets the Expander that is of type <typeparamref name="TExpander"/>.
20      /// </summary>
21      TExpander Expander { get; }
22  }
```

LISTING 2.3: The *IExpanderHandlerInteractor*

```
1  /// <summary>
2  /// Specifice an interface for an object that needs to be able to execute
       commands.
3  /// </summary>
4  public interface IExecutionInteractor
5  {
6      /// <summary>
7      /// Gets a value indicating whether the handler should be executed.
8      /// </summary>
9      bool CanExecute { get; }
10
11     /// <summary>
12     /// Executes the handler.
13     /// </summary>
14     void Execute();
15 }
```

LISTING 2.4: The *IExecutionInteractor*

The fact that *IExpanderHandlerInteractor* (Koks 2023m) derives from *IExecutionInteractor* (Koks 2023l) is another manifestation of OCP. This design decision allows for object types that need to be treated as executables by the CodeGeneratorInteractor. Examples are *RegionHarvesterInteractor* (Koks 2023u), *RegionRejuvenatorInteractor* (Koks 2023v), *PreProcessorInteractor* (Koks 2023t) and *PostProcessorInteractor* (Koks 2023s).

Listing 4.1 shows the CodeGeneratorInteractor that cohesively executes all of the IExecutionInteractor in order. The software engineer only has to focus on implementing the specific type of IExecutionInteractor without having to affect the implementation. This is by definition an example of "open for extension" and "closed for modifications".

The Liskov Substitution Principle

The LSP is a fundamental concept in object-oriented programming that deals with the behavior of derived objects (aka sub-types). The principal is named after Barbara Liskov, who first introduced the principle in a paper she co-authored in 1987. Barbara Liskov wrote the following statement as a way of defining subtypes (Robert C. Martin 2018, p. 95).

Liskov Substitution Principle

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

In simpler terms: An object *Volvo* of type *Vehical* should be able to be substituted for an object *Toyota* of type *Vehical* in any program that was defined in terms of *Vehical*, without affecting the program's correctness. This applies to all programs, not just a specific one.

The principle is based on the idea that a subtype should be semantically substitutable for its base type. This means that the subtype should behave in a way that is consistent with the expectations of the base type and should not introduce any new behaviors or violate any of the constraints imposed by the base type.

The practical implications of LSP are many. In software design, we should strive to create subtypes that are as similar as possible to their base types in terms of their behavior and the constraints they impose. In testing, it means that we should test subtypes to ensure that they behave correctly when used in place of their base types.

Consider *AbstractExpander* (Koks 2023a). This (abstract) object type allows for multiple implementations of *Expanders*. The primary example is the *CleanArchitectureExpander* (Koks 2023d) which is responsible for generating the expanded artifact that is part of this research. Different types of expanders could be added to the generator, ensuring they all behave in the same way.

The *ICreateGateway* (Koks 2023k) in Listing 2.5 is another example. The artifact has two implementations of this interface. The data of the entities are currently stored in the database, but harvest data is serialized to XML using the same ICreateGateway interface. With this design decision, it is very easy to adapt to a different type of storage mechanism if future requirement demand such a change.

One might notice the similarities with the OCP. The difference is that the OCP focuses on the extensibility of the system, without having to modify existing code. LSP ensures that the behavior of different subtypes is following the required functionality. LSP supports OCP, but it is not the only way of doing so.

```
namespace LiquidVisions.PanthaRhei.Generator.Domain.Gateways
{
    public interface ICreateGateway<in TEntity>
        where TEntity : class
    {
        bool Create(TEntity entity);
    }
}
```

LISTING 2.5: The *ICreateGateway*

```
internal class GenericRepository<TEntity> : ICreateGateway<TEntity>, // ... other interfaces
    where TEntity : class
{
    // ... other code

    public bool Create(TEntity entity)
    {
        context.Set<TEntity>().Add(entity);
        context.Entry(entity).State = EntityState.Added;

        return context.SaveChanges() >= 0;
    }

    // ... other code
}

internal class HarvestRepository : ICreateGateway<Harvest>, // ... other interfaces
{
```

```
    // .. other code

    public bool Create(Harvest entity)
    {
        if(string.IsNullOrEmpty(entity.HarvestType))
        {
            throw new InvalidProgramException("Expected harvest type.");
        }

        string fullPath = Path.Combine(
            parameters.HarvestFolder,
            app.FullName,
            $"{file.GetFileNameWithoutExtension(entity.Path)}.{entity.
                HarvestType}");

        serializer.Serialize(entity, fullPath);

        return true;
    }
}
```

LISTING 2.6: The *GenericRepository* and *HarvestRepository*

The Interface Segregation Principle

The ISP suggests that software components should have narrow, specific interfaces rather than broad, general-purpose ones. The ISP states that no client code should be forced 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 that are relevant to the clients that use them. This allows clients to use only the methods they need, without being forced to implement or depend on unnecessary methods (Robert C. Martin 2018, p. 104).

Overall, the ISP is about designing interfaces that are tailored to the specific needs of the clients that use them, rather than trying to create one-size-fits-all interfaces that may be bloated or unwieldy.

Take a look at Listing 2.7. In order to comply with the ISP, the design decision was made to separate all Creat, Read, Update, Delete (CRUD) operations into separate interfaces. In the example of the *AppSeederInteractor* (Koks 2023c) (see 2.7) only the delete and create gateways where required. An alternative approach was to create an IGateway interface containing all of the CRUD operations. Following this approach would lead to dependencies to all CRUD operations in the AppSeederInteractor.

```
// Create
public interface ICreateGateway<in TEntity>
    where TEntity : class
{
    bool Create(TEntity entity);
}

// Read
public interface IGetGateway<out TEntity>
    where TEntity : class
{
    IEnumerable<TEntity> GetAll();

    TEntity GetById(object id);
}

// Update
public interface IUpdateGateway<in TEntity>
    where TEntity : class
{
    bool Update(TEntity entity);
```

```
}

// Delete
public interface IDeleteGateway<in TEntity>
    where TEntity : class
{
    bool Delete(TEntity entity);

    bool DeleteAll();

    bool DeleteById(object id);
}

internal class AppSeederInteractor : IEntitySeederInteractor<App>
{
    private readonly ICreateGateway<App> createGateway;
    private readonly IDeleteGateway<App> deleteGateway;
    private readonly Parameters parameters;

    public AppSeederInteractor(IDependencyFactoryInteractor dependencyFactory)
    {
        createGateway = dependencyFactory.Get<ICreateGateway<App>>();
        deleteGateway = dependencyFactory.Get<IDeleteGateway<App>>();
        parameters = dependencyFactory.Get<Parameters>();
    }

    public int SeedOrder => 1;

    public int ResetOrder => 1;

    public void Seed(App app)
    {
        app.Id = parameters.AppId;
        app.Name = "PanthaRhei.Generated";
        app.FullName = "LiquidVisions.PanthaRhei.Generated";

        createGateway.Create(app);
    }

    public void Reset() => deleteGateway.DeleteAll();
}
```

LISTING 2.7: The Gateways for Create, Read, Update, Delete operations

lain how dependency ction can benefit the lementation and align n 5.5

The Dependency Inversion Principle

The DIP prescribes that high-level modules should not depend on low-level modules, and that both should depend on abstractions. The principle emphasizes that the architecture should be designed in such a way that the flow of control between the different objects, layers and components are 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 the way the data is stored or presented to the end user. Additionally, both the high-level and low-level implementations should only depend on abstractions or interfaces that define a contract for how they should interact with each other (Robert C. Martin 2018, p. 109).

This approach allows for great flexibility and a modular architecture. Modifications in the low-level implementations will not affect the high-level implementations as long as they still adhere to the contract defined by the abstractions and interfaces. Similarly, changes to the high-level modules will not affect the low-level modules as

long as they still fulfill the contract. This reduces coupling and ensures the evolvability system over time, as changes can be made to specific modules without affecting the rest of the system.

Manifestations in the artifacts are ample. One of which is the consistent use of the Dependency Injection pattern. In order to prevent the risks of displacing and dispersing dependencies all over the system (Mannaert et al. 2016, p. 214) we are using dependency containers. Each module is maintaining its own dependencies, which are bootstrapped at application startup (see Listing 2.8) (Koks 2023x).

```
// ... other code
cmd.OnExecute(() =>
{
    var provider = new ServiceCollection()
        .AddConsole()
        .AddDomainLayer()
        .AddApplicationLayer()
        .AddEntityFrameworkLayer()
        .AddInfrastructureLayer()
        .BuildServiceProvider();

    // ... other code
});

// ... other code
```

LISTING 2.8: Bootstrapping the dependencies of each component/layer of the generator artifact.

A more abstract example is the separation of required modules into separate component libraries. This applies to both the generated and generator artifact (see Figure 2.3). The actual compliance to the DIP is how the flow of control between the components is organized. This is accurately depicted in Figure 2.4 modularity.

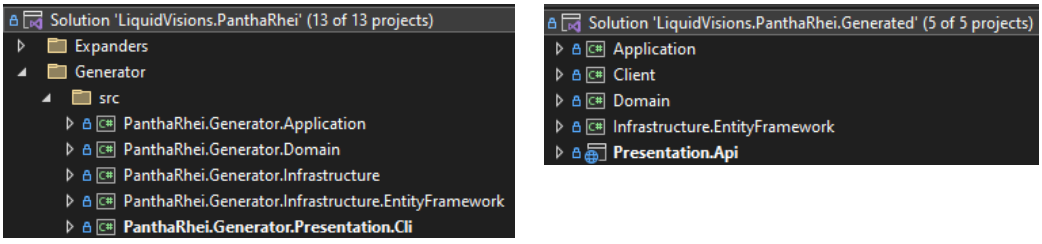


FIGURE 2.3: Separation of component libraries.

2.2.2 Layers and components

CA organizes software systems into distinct layers or components, each with its own responsibilities. This structure promotes the separation of concerns, maintainability, testability, and adaptability. The following section is a short description of each of the layers (Robert C. Martin 2018).

Domain layer

This layer contains the core business objects, rules, and domain logic of the application. 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 that orchestrate the interaction between entities and external systems. Use cases define the behavior of the application in terms of the actions users can perform and the expected outcomes. This layer is responsible for coordinating the flow of data between the domain layer and the presentation or infrastructure layers, while remaining agnostic to the specifics of the user interface or external dependencies.

Presentation layer

This layer is responsible for translating data and interactions between the use cases and external actors, such as users or external systems. Interface adapters include components like 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 any configuration or initialization code required to set up the system's runtime environment.

By organizing code into these layers and adhering to the principles of CA, developers can create software systems that are more flexible, maintainable, and testable, with well-defined boundaries and separation of concerns.

2.2.3 The Design Elements

In the context of NS approach, the goal is to design a software system that is highly modular, maintainable and testable. The accumulation of the Desing principles discussed in chapter 2.2.1 leads to the following generalization of the architecture. Each of the following elements has a crucial role to achieve the design goals.

Entities

Entities are the core business objects of the application, representing the fundamental concepts and rules of the domain. They encapsulate the data and behavior that are essential to the application's functionality.

Interactors

Interactors, also known as Use cases, encapsulate the application's business logic and represent specific actions that can be performed by the system. They are responsible for coordinating the work of other components and ensuring that the system behaves correctly.

RequestModels

RequestModels are used to represent the data required by a specific interactor. They provide a clear and concise representation of the data required by the Use Case, making it easier to manage and modify the application.

ViewModels

ViewModels are part of the presentation layer and are responsible for managing the state of the user interface. They receive data from the Presenters and update the user interface accordingly. They are also responsible for handling user input and sending it to the Controllers for processing.

Controllers

Controllers are responsible for handling requests from the user interface and routing them to the appropriate Interactor. They are typically part of the user interface layer and are responsible for coordinating the work of other components.

Presenters

Presenters are responsible for formatting and presenting data to the user interface. They receive data from the Interactor and convert it into a format that can be easily displayed to the user. They are also responsible for handling user input and sending it back to the Interactor for processing.

Gateways

A *Gateway* provides an abstraction layer between the application and its external dependencies, such as databases, web services, or other systems. They allow the system to be decoupled from its external dependencies and can be easily replaced or adapted if needed.

Boundaries

A *Boundary* refers to an interface or abstraction that separates different layers or components of a system. The purpose of these boundaries is to promote modularity, evolvability and testability by enforcing the separation of concerns, allowing each layer to evolve independently.

2.2.4 The Dependency rule

An essential aspect is described as the dependency rule. The rule has been stated as followed (Robert C. Martin 2018, p. 206).

> **The flow of control**
>
> *Source code dependencies must point only inward, toward higher-level policies*

The flow of control is intended to follow the DIP and can be represented schematically as concentric circles containing all the components described in chapter 2.2.2. The arrows in Figure 2.4 clearly show that the dependencies flow from the outer

layers to the inner layers. This ensures that the domain logic can evolve independently from external dependencies or certain specific technologies. Additionally, it separated the application and domain logic from how it is presented to the user.

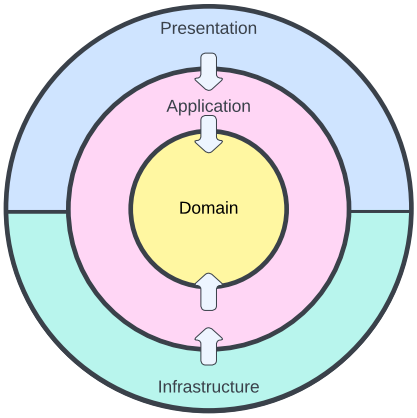


FIGURE 2.4: Flow of control

3 Requirements

In this chapter, we will explain that the artifact requirements are composed of two distinct parts. The research aims to examine the level of convergence between CA and NS. As such, the design and architecture requirements are intentionally, and entirely based on the principles and design approach of CA. A full description of this requirement can be found in section 3.2. In order to analyze the level of convergence, the artifacts need to comply with the evolvability requirements of the theory of NS, described in section 3.1 of this chapter.

3.1 Research requirements

In order to address the research objectives we outlined in Chapter 1.1, we need to know what violations in the context of NS can occur in a software artifact. In chapters 2.1 we have shown that NS attempts to achieve evolvability by achieving stability. Through the extensive and consistent use of the NS theorems, a modular design emerges that is free of instabilities. Alternatively, in terms used by the NS theorem: the absence of combinatorial effects. However, as described in section 3.2, the artifacts are designed based on the principles and design approach of CA. The NS theorems are not considered using the design phase of the artifacts.

To be able to analyze the stability of the artifacts the following functional requirement specifications are reused (Mannaert et al. 2016, pp. 254–259). They are noted without the mathematical formulas that are present in cited sources.

Research Requirement 1

An information system needs to be able to represent instances of data entities. A data entity consists of several data fields. Such a field may be a basic data field representing a value of a reference to another data entity.

Research Requirement 2

An information system needs to be able to execute processing actions on instances of data entities. A processing action consists of several consecutive processing tasks. Such a task may be a basic task, i.e., a unit of processing that can change independently, or an invocation of another processing action.

Research Requirement 3

An information system needs to be able to input or output values of instances of data entities through connectors.

Research Requirement 4

An existing information system representing a set of data entities, needs to be able to represent:

- *a new version of a data entity that corresponds to including an additional data field*
- *an additional data entity*

Research Requirement 5

An existing information system providing a set of processing actions, needs to be able to provide:

- *a new version of a processing task, whose use may be mandatory*
- *a new version of a processing action, whose use may be mandatory*
- *an additional processing task*
- *an additional processing action*

3.2 Artifact requirements

The artifacts adhere to the design approach of CA, described in chapter 2.2, as much as possible. In order to do so the following requirements are to be applied.

3.2.1 Artifact components

In 2.2.2 Layers and components we describe that in a typical application, one of the goals of CA is to separate the domain logic from Impl details like (user) interface or storage mechanisms. Therefore, the artifact must follow the layered architecture of CA (see figure 2.4)

In the case of the Designing and Creating the generator artifact there are two separate infrastructure layers. The first one has a dependency on EntityFrameworkCore[1] technology for data persistence in an Azure SQL database. The other one handles persistence by serialization of data on a windows machine.

3.2.2 Flow of control

One of the SOLID design principles is the Dependency Inversion Principle (DIP). This principle affects the. As described in 2.2.4 The Dependency rule, this affects the flow of control of the artifacts components. To achieve the evolvability of each of the components, the flow of control must adhere to the following statement. This is also depicted in Figure 2.4.

The component dependencies must point only inward, toward higher-level policies. To make it more strict, the Presentation and infrastructure layers only depend on the Application layer The Application layer only depends on the Domain Layer.

[1] `https://learn.microsoft.com/en-us/ef/core/`

3.2.3 Adherence to Design principles

In 2.2.1 The Design principles we describe what needs to be done to adhere to the SOLID design principles. Each design pattern that is applied to the architecture of artifacts adheres at least to one of the SOLID principles.

3.2.4 Screaming Architecture

The essence of Screaming Architecture is to design a system where the structure and purpose of each component and layer are immediately apparent to anyone looking at it. This is achieved through clear and consistent naming conventions of the classes, namespaces and, components.

In the literature of CA, there is no mention of rules or recommendations for naming conventions. The most commonly practiced naming convention of nouns (for data entities) and verbs (for actions) will be used. This is also advocated by the literature of NS (Mannaert et al. 2016, p. 357).

Table 3.1 lists the required naming convention for the different layers, whereas table 3.2 lists the naming convention of the recurring elements. For more information about these elements see 2.2.3.

Component	**Filename**	**Namespace**
Domain	[Prod].Domain	[Company].[Prod].Domain
Application	[Prod].Application	[Company].[Prod].Application
Presentation	[Prod].Presentation.[Tech]	[Company].[Prod].Presentation.[Tech]
Infrastructure	[Prod].Infrastructure.[Tech]	[Company].[Prod].Infrastructure.[Tech]

TABLE 3.1: Naming convention components

Component	**Element type**	**Naming Convention**
Presentation	Controller Impl	*Noun*Controller
	ViewModelMapper Impl	*Noun*ViewModelMapper
	Presenter Impl	*VerbNoun*Presenter
	ViewModel Impl	*Noun*ViewModel
	DI Bootstrapper	DependencyInjectionBootstrapper
Application	Boundary Impl	*VerbNoun*Boundary
	Boundary interface	IBoundary
	Gateway interface	I*Verb*Gateway
	Interactor interface	I*Verb*Interactor
	Interactor Impl	*VerbNoun*Interactor
	Mapper Interface	IMapper
	RequestModelMapper Impl	*VerbNoun*RequestModelMapper
	Presenter Interface	IPresenter
	Validator Interface	IValidator
	Validator Impl	*VerbNoun*Validator
	DI Bootstrapper	DependencyInjectionBootstrapper
Infrastructure	Gateway Impl	*Noun*Repository
	DI Bootstrapper	DependencyInjectionBootstrapper
Domain	Data Entity Impl	*Noun*
	DI Bootstrapper	DependencyInjectionBootstrapper

TABLE 3.2: Naming convention of recurring elements

3.2.5 Recurring elements

For each data entity defined in the model (see chapter 4.2.1), a fixed set of required elements, listed in table 3.2 will be added to the appropriate component, to comply with the design of the architecture.

uirement mbt code ex-
sion toevoegen (pag

4 Designing artifacts.

4.1 Designing and Creating the generator artifact

In the previous chapters, we discussed the importance of software stability and evolvability in order to cope with the continuously changing business and technological requirements. Although considered from a much broader perspective, the Greek philosopher *Heraclitus* described this state of constant *flux* with his famous statement *Pantha Rhei*. His statement was the main inspiration for the name of the generator artifact.

The following sections discuss the design, implementations and functionality of the Pantha Rhei, the generator artifact. The artifact is created in fulfillment of this research. Whereas the name of the artifact was inspired by Heraclitus, the functional idea behind the artifact was inspired by the theory behind NS and by the Prime Radiant application of the company NSX [1]. The concept of code generation in order to achieve stable and evolvable software is one of the important aspects of NS.

For the interested readers of this thesis, it is rather simple to install the Pantha Rhei application by following the instruction in the appendix B Installing & using Pantha Rhei.

4.1.1 The Flux command

4.1.2 Expanders

As described in section 4.1, and portraited in figure 4.1, expanders are used as plugins by the Pantha Rhei expander artifact. There are a couple of prerequisites applicable before the expander can be dynamically loaded as a plugin at runtime.

Prerequisite 1: Project dependency

The expander should have a dependency on the Dynamic Link Library (DLL) of the project *PanthaRheiGeneratorDomain* (Koks 2023r).

Prerequisite 2: Implements IExpanderInteractor

In order to behave like an expander, one should behave like an expander. This is done by implementing the *IExpanderInteractor* (Koks 2023n) interface. Although not required, it is strongly recommended to use the abstract *AbstractExpander* (Koks 2023a) as it contains all the routines required implementations of *IExecutionInteractor* (Koks 2023l) like Harvesters, Rejuvenators, Pre-, and PostProcessors.

[1] `https://normalizedsystems.org/prime-radiant/`

Prerequisite 3: Implements AbstractExpanderDependencyManagerInteractor

The Dependency Injection pattern is an exciting and beneficiary pattern that entirely adheres to the DIP principle. By implementing the abstract *AbstractExpanderDependencyManagerInteractor* (Koks 2023b) all, for the expander specific implementations of the following interfaces will automatically be registered and made available for runtime processing.

- The expander which should be an implementation of *IExpanderInteractor* (Koks 2023n) or *AbstractExpander* (Koks 2023a)
- When applicable, the expander handers, which should be an implementation of *IExpanderHandlerInteractor* (Koks 2023m)
- The default *RegionHarvesterInteractor* (Koks 2023u) will automatically be executed during the generation process.
- Specific implementations of Harversters, which should be an implementation of *IHarvesterInteractor* (Koks 2023o)
- The default *RegionRejuvenatorInteractor* (Koks 2023v) will automatically be executed during the generation process.
- Specific implementations of Rejuvenators, which should be an implementation of *IRejuvenatorInteractor* (Koks 2023q)
- The default PostProcessor *InstallDotNetTemplateInteractor* (Koks 2023p) will automatically be executed during the generation process.
- Specific implementation of PostProcessors, which should be an implementation of *PostProcessorInteractor* (Koks 2023s)
- The default PreProcessor *UnInstallDotNetTemplateInteractor* (Koks 2023y) will automatically be executed during the generation process.
- Specific implementation of PreProcessors, which should be an implementation of *PreProcessorInteractor* (Koks 2023t)

4.1.3 Plugin Architecture

When all preconditions are met and the expander is compiled, the expander consists of a DLL and a set of templates. The Generator artifact considers the expanders as optional plugins, which are dynamically loaded at runtime, through a method called assembly-binding. See section 4.1.2 Expanders for a full explanation of the required preconditions.

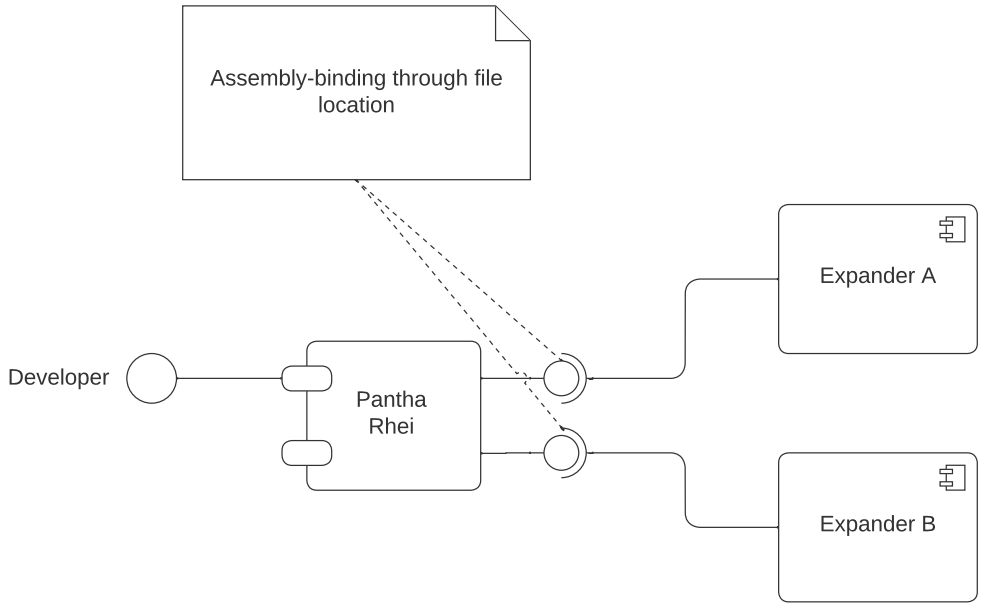


FIGURE 4.1: Expanders are considered plugins

By implementing the expanders as plugins, the design adheres to several SOLID principles. First and foremost, SRP is respected because an expander should generate one, and precisely one construct (Mannaert et al. 2016, p. 403). In the case of this research, this construct is an application following the design and principles of the design approach of CA with a Restful API interface. Furthermore, it supports the OCP principles because developers can introduce a new version of the expander as a separate plugin if this is required. LSP. By extension, this also adheres to the LSP principle, as expanders can be replaced with other implementations of expanders, without affecting the rest of the application.

4.1.4 The executer object

A vital implementation that facilitates a high degree of cohesion, whilst maintaining low coupling and adhering to the SRP principle is the use of the *IExecutionInteractor* (Koks 2023l) interface. The generation process is designed to execute "tasks" in a predefined order. By using the IExecutionInteractor it is possible to design each of the tasks as separate classes, entirely complying, or enabling all of the SOLID principles. The

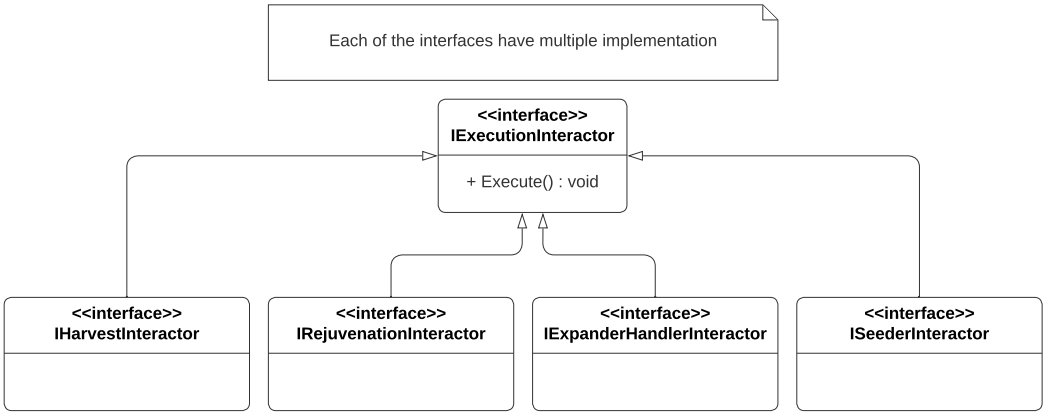


FIGURE 4.2: Both high cohesion and low coupling by using the IExecutionInteractor

As depicted in listing 4.1, this design leads to a cohesive design where all tasks are gracefully executed from a single point in the application.

```
/// <summary>
/// Implements the contract <seealso cref="ICodeGeneratorInteractor"/>.
/// </summary>
internal sealed class CodeGeneratorInteractor : ICodeGeneratorInteractor
{
    // ... other code

    /// <inheritdoc/>
    public void Execute()
    {
        foreach (IExpanderInteractor expander in expanders
            .OrderBy(x => x.Model.Order))
        {
            expander.Harvest();

            Clean();

            expander.PreProcess();
            expander.Expand();
            expander.Rejuvenate();
            expander.PostProcess();
        }
    }

    // ... other code
}
```

LISTING 4.1: The *CodeGeneratorInteractor*

4.1.5 The meta-model

The meta-model is a higher-level abstraction of the model that is used to describe the characteristics of a software system in the context of this research. It contains the essential characteristics that are needed in order to generate software applications and components. It contains the structures, entities and relationships within the Generated Artifact. Having a meta-model ensures a standardized way to describe and define the software components and their interactions, enabling the design and analysis of modular, evolvable, and scalable software systems.

In this context, the meta-model serves as a blueprint for the software system, describing its components, such as entities, fields, relationships, and expanders, along

with their attributes and constraints. The meta-model is used to generate the Generated Artifact (4.2). Moreover, it is potentially reusable for other types of applications, which should lead to the same characteristics of stability and evolvability as the one used for this research.

The following sections describe all the entities that are part of the meta-model. These entities are the basis of the Entity Relation Diagram (ERD) depicted in Appendix A. The ERD is the official model representing the meta-model.

The App entity

The App entity represents an application and is regarded as the highest entry point for the Generator artifact. The App Entity and subsequent entities contain all the information needed to generate the Generated Artifact described in section 4.2.

Name	DataType	Description
Id	Guid	Unique identifier of the application
Name	string	Name of the application
FullName	string	Full name of the application
Expanders	List of Expanders	The Expanders that will be used during the generation process.
Entities	List of Entities	The Entities that are applicable for the Generated artifact.
ConnectionStrings	List of ConnectionStrings	The ConnectionString to the database that is used by the Generator Artifact.

TABLE 4.1: The fields of the App entity

The Component entity

The Component entity represents a software component that can be part of an application. Based on this entity the Generator Artifact can make design time on where to place certain elements

Name	DataType	Description
Id	Guid	Unique identifier of the component
Name	string	Name of the component
Description	string	Description of the component
Packages	List of Package	The Packages that should be applied to the component.
Expander	Expander	Navigation property to the Expander entity.

TABLE 4.2: The fields of the Component entity

The ConnectionString entity

The ConnectionString entity represents a ConnectionString used by an application to connect to a database or other external system.

Name	DataType	Description
Id	Guid	Unique identifier of the ConnectionString
Name	string	Name of the ConnectionString
Definition	string	Definition of the ConnectionString
App	App	Navigation property to the App entity

TABLE 4.3: The fields of the ConnectionString entity

The Entity entity

The Entity entity represents an entity in the application's data model.

Name	DataType	Description
Id	Guid	Unique identifier of the entity
Name	string	Name of the entity
Callsite	string	The source code location where the entity is defined. In the case of a C# artifact, this is to determine the name of the namespace.
Type	string	Type of the entity
Modifier	string	Modifier of the entity (e.g. public, private)
Behavior	string	The behavior of the entity (e.g. abstract, virtual)
App	App	Navigation property to the App entity.
Fields	List of Fields	The Fields property represents a collection of the fields that make up the entity.
ReferencedIn	List of Fields	Represents a navigation property to a Field that uses the current entity as a return type.
Relations	List of Relationships	List of relationships involving this entity
IsForeignEntityOf	List of Relationships	List of relationships where this entity is the foreign entity

TABLE 4.4: The fields of the Entity entity

The Expander entity

The Expander entity represents an expander, which is responsible for generating code for an application. The Generator Artifact attempts to execute all expanders that are related to the selected App.

Name	DataType	Description
Id	Guid	Unique identifier of the expander
Name	string	Name of the expander
TemplateFolder	string	relative path to the templates that are used by the expander.
Order	int	The order in which the expander is executed
Apps	List of Apps	List of applications associated with the expander.
Components	List of Components	List of components associated with the expander

TABLE 4.5: The fields of the Expander entity

The Field entity

The Field entity represents a field or property of an entity in an application's data model. Each field has a unique ID, name, and other properties such as its return type, modifiers, and behavior. It can be associated with an entity and can have relationships with other entities. The IsKey and IsIndex properties indicate whether the field is part of the primary key or an index of the entity, respectively.

Name	DataType	Description
Id	Guid	Unique identifier of the field
Name	string	Name of the field
ReturnType	string	Return type of the field
IsCollection	bool	Whether the field is a collection or not
Modifier	string	Modifier of the field (e.g. public, private)
GetModifier	string	Modifier of the get accessor for the field
SetModifier	string	Modifier of the set accessor for the field
Behavior	string	The behavior of the field (e.g. abstract, virtual)
Order	int	The order of the field within its entity
Size	int?	The size of the field
Required	bool	Whether the field is required or not
Reference	Entity	The entity that this field refers to
Entity	Entity	A navigation property to the parent entity
IsKey	bool	Indicates whether the field is part of the primary key
IsIndex	bool	Indicates whether the field is part of an index
RelationshipKeys	List of Relationships	A List of entities that are defined as relations.
IsForeignEntityKeyOf	List of Relationships	List of relationships to the field that is the foreign key

TABLE 4.6: The fields of the Field entity

The Package entity

The Package entity represents a software package that can be used by a component. This could either be a Nuget package in the case of .NET projects, or for example npm packages for web projects.

Name	DataType	Description
Id	Guid	Unique identifier of the package
Name	string	Name of the package
Version	string	Version of the package used
Component	Component	Component associated with the package

TABLE 4.7: The fields of the Package entity

The Relationship entity

The Relationship entity represents a relationship between two entities in the App's data model. The Relationship entity has proper cardinality support. Relationships are bidirectional and can be navigated from either entity.

Name	DataType	Description
Id	Guid	Unique identifier of the relationship
Key	Field	The key field of the relationship
Entity	Entity	Navigation property to the parent Entity
Cardinality	string	The cardinality of the relationship
WithForeignEntityKey	Field	The foreign key field of the relationship, pointing to a Field entity.
WithForeignEntity	Entity	The entity associated with the foreign key field
WithCardinality	string	The cardinality of the relationship with the foreign entity
Required	bool	indicates whether the relationship is required or not

TABLE 4.8: The fields of the Relationship entity

4.2 Desigingin and generating a generated artifact

4.2.1 The model

4.3 Research Design Decision

In accordance to chapter 3.1, the following design decisions have been made to be able to analyze the stability and the evolvability of both of the artifacts.

4.3.1 Fulfilling Research Requirement 1

The following Data entities will be created as part of the initial versions of both artifacts. A full description of the data entities and their relations towards each other can be reviewed in chapter 4.1.5 The meta-model. A description of all attributes are included.

oegen dat de entity,
entiteiten een relaties
t krijgen met compo-
t entitiet zodat plaatsen
entiteiten in componen-
niet meer in designtime
t.

5 Evaluation results

5.1 Converging principles

In order to address the goal of this research outlined in Chapter 1.1, a comprehensive analysis of the existing literature on the SOLID principles and NS theorems have been conducted. Furthermore, the development of the artifact has provided valuable insights into this subject matter.

In the following sections, a systematic cross-referencing approach has been applied to assess the level of convergence between each of the SOLID principles of CA and the NS theorems. Along with a brief explanation, the level of convergence is denoted as follows:

Fully converges	++	This indicates a high degree of alignment between the respective SOLID principle and NS theorem. The application of either principle or theorem results in a similar impact on the software design.
Supports convergence	+	In this case, the SOLID principle assists in implementing the NS theorem through specific design choices. However, it is essential to note that applying the principle does not inherently ensure adherence to the corresponding theorem.
No convergence	−	This denotation signifies a lack of alignment between the SOLID principle and the corresponding theorem.

5.1.1 Converging the Single Responsibility Principle

SOC	++	SRP and SOC share a common objective: facilitating evolvable software systems through the promotion of modularity, low coupling, and high cohesion. While there may be some differences in granularity when applying both principles according to the original definition of SOC, the more stringent definition of Separation of Concerns, offered by the NS Theorems 2.1.7 minimizes these differences. As a result, the two principles can be regarded as practically interchangeable. In conclusion, SRP and SOC exhibit full convergence, as they both emphasize encapsulating 'responsibilities' or 'concerns' within modular components of a software system.
DvT	+	While not immediately apparent, SRP offers supports for the DvT theorem. While SRP emphasizes limiting the responsibility of each module, it does not explicitly require handling changes in data structures. However, following glssrp can still indirectly contribute to achieving DvT by promoting the Law of Demeter [1], which encourages modules to interact with each other only through well-defined interfaces. This approach can minimize the impact of data structure changes, although it does not guarantee full convergence with DvT.
AvT	+	Although not that apparent, SRP supports the DvT theorem. While SRP emphasizes limiting the responsibility of each module, it does not explicitly require handling changes in data structures. However, following glssrp can still indirectly contribute to achieving DvT by promoting the Law of Demeter [2], which encourages modules to interact with each other only through well-defined interfaces. This approach can minimize the impact of data structure changes, although it does not guarantee full convergence with DvT.
SOS	−	The convergence between SRP and the SOS theorem is not as direct as with other theorems. SRP focuses on assigning a single responsibility to each module but does not explicitly address state management. Nevertheless, by following SRP, developers can create modules that manage their state, which indirectly contributes to SOS.

TABLE 5.1: Converge SRP with NS

5.1.2 Converging the Open/Closed Principle

SOC	++	The OCP converges with the SOC theorem. OCP states that software implementations should be open for extension but closed for modification. When applying OCP correctly, modifications are separated from the original implementations. For example by creating a new implementation of an interface or a base class. Conversely, adhering to SOC does not guarantee the fulfillment of OCP, as SOC focuses on modularization and encapsulation, rather than the extensibility of modules.
DvT	+	The OCP supports the DvT theorem. While DvT aims to handle changes in data structures without impacting the system, OCP focuses on the extensibility of software entities. OCP does not explicitly address data versioning, and thus does not guarantee full convergence with DvT. However, by designing modules that follow OCP, developers can create components that are more adaptable to changes in data structures.
AvT	++	The OCP converges with the AvT theorem. Both principles emphasize the importance of allowing changes or extensions to actions or operations without modifying existing implementations. By adhering to OCP, developers can create modules that can be extended to accommodate new actions or changes in existing ones, effectively achieving AvT.
SOS	−	The OCP has an indirect relationship with the Separation of States (SoSt) theorem. However, not on a level where we can speak of convergence. SOS emphasizes isolating different states within a system. Adhering to OCP alone does not guarantee full separation of states.

TABLE 5.2: Converge OCP with NS

5.1.3 Converging the Liskov Substitution Principle

SOC	++	Adhering to LSP in the software design leads to a more modular design and separation of specific concerns. Therefore we state that LSP converges with SOC. LSP states that objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program. This can only be achieved by a strict separation of concerns in combination with Action version Transparent implementations of the signature.
DvT	−	LSP has a limited alignment with the DvT theorem. While LSP focuses on the substitutability of objects in class hierarchies, DvT aims to handle changes in data structures without impacting the system. By following LSP, developers can ensure that derived classes can be substituted for their base classes, which may help reduce the impact of data structure changes on the system. However, LSP does not explicitly address data versioning and thus does not guarantee full convergence with DvT.
AvT	+	The LSP supports the AvT theorem. Both principles emphasize the importance of allowing the extensibility of the system, without negatively impacting the desired requirements. By adhering to LSP, developers can create class hierarchies that can be easily extended to accommodate new actions or changes in existing ones, which may contribute to achieving AvT. However, adhering to LSP alone may not guarantee full convergence with AvT.
SOS	−	By designing class hierarchies according to LSP, developers can create components that are less prone to side effects caused by shared states. However, the alignment between LSP and SOS is very weak, and adhering to LSP alone may not guarantee full separation of states.

TABLE 5.3: Converge LSP with NS

5.1.4 Converging the Interface Segregation Principle

SOC	++	The ISP converges with the SOC theorem, as both principles emphasize the importance of modularity and the separation of concerns. ISP states that clients should not be forced to depend on implementation they do not use, promoting the creation of smaller, focused interfaces. By adhering to SOC and designing target interfaces, inherently support SOC, leading to more evolvable software systems.
DvT	+	The ISP has an indirect relationship with the DvT theorem. When adhering to the ISP, a developer can create a specific interface supporting a specific version of data entities. Although this approach can help minimize the impact of data structure changes on the system, it does not guarantee full DvT information system.
AvT	+	The ISP supports the AvT theorem. Both principles emphasize the importance of separating actions within a system. ISP promotes the creation of interfaces for each (version of an) action, which aligns with the core concept of AvT. This alignment allows for a more manageable system, where changes in one action do not lead to ripple effects throughout the entire system. Although the adherence to ISP does not guarantee full compliance with the AvT
SOS	−	There is no alignment between ISP and SOS. Mostly because ISP focuses on the separation of interfaces and abstract classes, while SOS emphasizes isolating different states within a system. This is an implementation concern. By no means the application of ISP could lead to a separation of state.

TABLE 5.4: Converge ISP with NS

5.1.5 Converging the Interface Segregation Principle

SOC	++	This principle converges with the SOC theorem. DIP states that high-level modules should not depend on low-level modules. When unavoidable, high-level modules should depend on abstractions of low-level modules and abstractions should not depend on details. By adhering to DIP correctly, developers can create modular and decoupled software systems, which aligns to break down a system into evolvable components. This convergence enables developers to create maintainable, scalable, and adaptable software systems that effectively manage complexity. Dependency Injection is a valuable (but not the only or mandatory) aspect of DIP. We have observed that the claim that the technique of Dependency Injection solves coupling between classes in an application is dangerous and in some cases wrong (Mannaert et al. 2016, p. 215). Nevertheless, this technique, when applied correctly (see 2.2.1), the artifact have pointed out that it has been a great asset in creating evolvable software, especially in the aspect of SOC.
DvT	+	Adhering to the DIP can indirectly support the DvT theorem. By adhering to the DIP, developers can promote implementations that encourage modules to interact with each other only through well-defined interfaces or abstractions. This approach can help minimize the impact of data structure changes on the system but does not guarantee full compliance with DvT.
AvT	+	The DIP can support the AvT theorem. Both principles emphasize the importance of isolating actions or operations within a system. By adhering to DIP, developers can create modular components that interact through abstractions, which may contribute to achieving AvT. However, the alignment between DIP and AvT less strong than with SOC, and adhering to DIP alone will not guarantee a system that entirely complies to AvT.
SOS	−	There is a very weak alignment between DIP and SOS. Although Developers can create components that are less prone to side effects caused by a shared state, this can hardly be contributed to the DIP. Therefore it is stated that there is no convergence between the two principles.

TABLE 5.5: Converge DIP with NS

5.2 Converging elements

Strong convergence	++	Both elements have a high level of similarity or are closely related in terms of their purpose, structure, or functionality.
some convergence	+	Both elements have some similarities or share certain aspects in their purpose, structure, or functionality, but they are not identical or directly interchangeable.
No convergence	−	The elements are not related or have no significant similarities in terms of their purpose, structure, or functionality.

5.2.1 Converging the Entity element

The following NS elements are directly convergent with the *Entity* element of CA. The *Task*, *Flow*, *Connector* and *Trigger* elements are in no way convergent as they differ in the perspective of structure, intent, purpose and functionality.

Data	++	Both the elements represent a domain object that is part of the ontology or data schema of the application. Additionally, They both contain information about the associated attributes and relationships.

TABLE 5.6: Converge CA *Entity* element with NS elements

5.2.2 Converging the RequestModel element

The following NS elements are convergent with the RequestModel element of CA. The Task, Flow, Connector and Trigger elements are in no way convergent as they differ in the perspective of structure, intent, purpose and functionality.

Data	+	The RequestModel and NS Data element convergent to some degree. Both are involved in defining the structure of data used in the system. This includes required information about attributes and relationships. RequestModels represent input data for a specific interactor and will most likely contain only the information that is needed for that specific use case. The RequestModel could also carry non Data element specific information that is required to execute the use case.

TABLE 5.7: Converge CA 'RequestModel element' with NS elements

5.2.3 Converging the ResponseModel element

The following NS elements are convergent with the ResponseModel element of CA. The Task, Flow, Connector and Trigger elements are in no way convergent as they differ in the perspective of structure, intent, purpose and functionality.

Data	+	The ResponseModel and NS Data element convergent to a limited degree. Both are involved in defining the structure of data used in the system. This includes required information about attributes and relationships. The ResponseModel represents output data for a specific interactor and will most likely contain only the information that is needed in response for that specific use case.

TABLE 5.8: Converge CA 'ResponseModel element' with NS elements

5.2.4 Converging the Presenter element

5.2.5 Converging the Element element

5.2.6 Converging the Interactor element

5.2.7 Converging the Boundary element

5.2.8 Converging the Controller element

The following NS elements are convergent with the *Controller* element of CA. The *Data*, *Task* and *Flow* elements are not convergent as they do not align in the perspective of structure, intent, purpose and functionality.

Connector	+	The *Controller* and NS *Connector* element are convergent to some degree. Both elements are involved in communication between components. The use of the Controller is a bit more strict it strictly defines communication from external parts of the systems, involving specific *Interactor*.
Trigger	++	The *Controller* and the *Trigger* element of NS are convergent to some degree as they both can initiate actions based on external events or requests. A *Controller* can respond to external requests and invoking the appropriate interactor.

TABLE 5.9: Converge CA 'Controller element' with NS elements

Evaluating the findings

	SoC	DVT	AVT	SoS
SRP	++	+	+	−
OCP	++	+	++	−
LSP	++	−	+	−
ISP	++	+	+	−
DIP	++	+	+	−

TABLE 5.10: Convergence between SOLID principles and NS theorems (abbreviated)

6 Discussion

7 Conclusions

- Ca offers structure, principles and guidelines on how to build something. On top of that, NST also offers guidelines in order to apply actual changes.

A The Entity Relationship Diagram of the Meta Mode

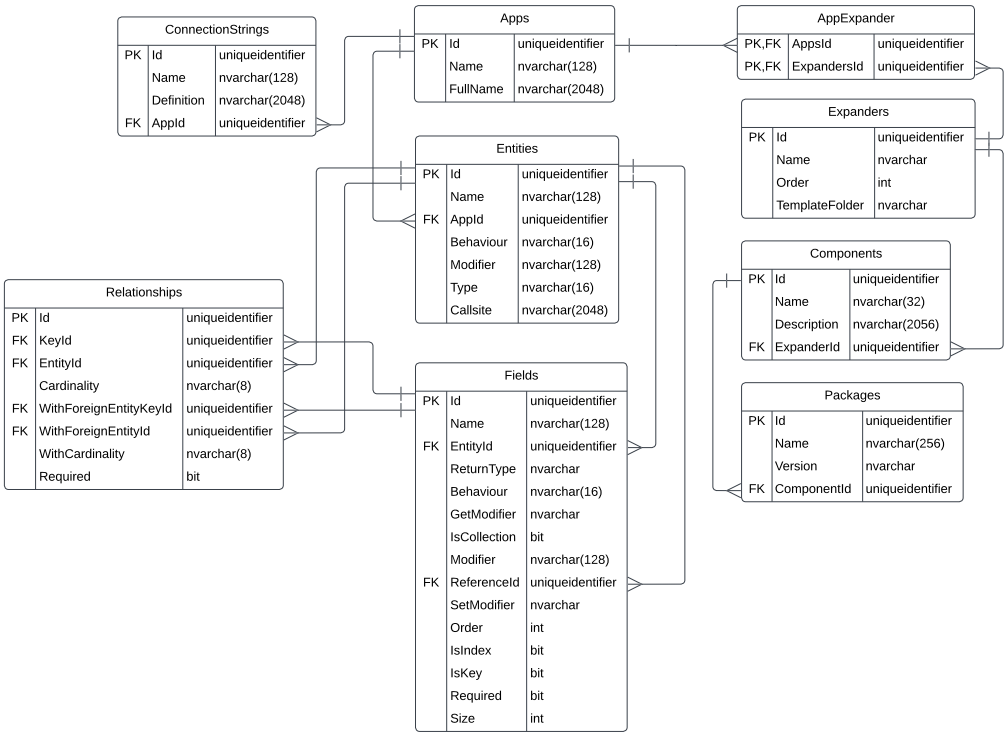


FIGURE A.1: Entity Relationship Diagram of the MetaModel

B Installing & using Pantha Rhei

B.1 Installation instructions

Step 1: Create output folder

Create an output folder so that the applications...

- ...has a location where to find the required expanders.
- ...has a location where the log files are stored.
- ...has a location where the result of the generation processes can be stored.

The location of the output folder is irrelevant.

Step 2: Create the Nuget configuration file

Add a configuration file named *nuget.config* file to the output folder with the the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <clear />
        <add key="github" value="https://nuget.pkg.github.com/liquidvisions/
            index.json" />
    </packageSources>
        <packageSourceCredentials>
        <github>
            <add key="Username" value="" />
            <add key="ClearTextPassword" value="{email me for a valid access
                token @ gerco.koks@outlook.com} " />
        </github>
    </packageSourceCredentials>
</configuration>
```

LISTING B.1: The content of the Nuget configuration file

This config file is needed for the following step where the Pantha Rhei application is installed. The file contains the information to the private feed where the Pantha Rhei application can be downloaded and installed.

Step 3: Installing Pantha Rhei

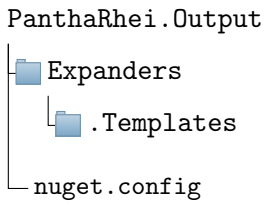
Open a console in on the location where the Nuget configuration file is stored. The following command will download the package, and start the installation process which is executed in the background.

```
dotnet tool install LiquidVisions.PanthaRhei.Flux -g
```

LISTING B.2: The install command

Step 4: Download & Install the Expanders

By clicking on the following link, an archived folder will be presented as a download by your browser. Download the archived folder and extract it on completion. Store the extracted folder, in a subfolder called *Expanders* in the root of the output folder. By doing so, the following folder structure should be available:



The Pantha Rhei application is now ready for use.

Step 5: Setup a SQL database

Currently, Pantha Rhei is working with an MS SQL database for storing the model of the applications. Set up a SQL Database. This can either be a licensed version of SQL Server, the free-to-use SQL Express or an Azure SQL instance. See `https://www.microsoft.com/en/sql-server/sql-server-downloads` for more information. Make sure to have a valid connection string to the SQL server instance that is needed in step B.1 Step 6: Execute the command.

Step 6: Execute the command

Pantha Rhei is used by executing the *flux* command with the parameters as described in table B.1

```
flux --root "{folder}"
     --mode Default
     --app "{uniqueid}"
     --db "{connectionstring}"
```

LISTING B.3: Example command executing Pantha Rhei

–root	A mandatory parameter that should contain the full path to the output directory B Installing & using Pantha Rhei.
–db	A mandatory parameter that contains the connection string to the database.
–app	A mandatory parameter indicating the unique identifier of the application that should be generated.
–mode	An optional parameter that determines if a handler should be executed. *Default* is the default fallback mode (see B.2).
–reseed	An optional parameter that bypasses the expanding process. The model will be thoroughly cleaned and reseeded based on the entities of the expander artifact. This enables to a certain extent the meta-circularity and enables the expander artifact to generate itself.

TABLE B.1: The *flux* command line parameters

RunModes are available to isolate the execution of the ExpanderHandler. It requires a current implementation shown in listing B.4. The following RunModes are available.

Default	This is the default generation mode that executes all configured handlers of the CleanArchitectureExpander. This will also install the required Visual Studio templates which are needed for scaffolding the Solution and C# Project files. Furthermore, it also executes the Harvest and Rejuvenation handlers. This mode will clean up the entire output folder prior after the Harvesting process is finished prior to the execution of the handlers.
Extend	This mode will skip the installation of the Visual Studio templates and the project scaffolding. It will not clean up the output folder but will overwrite any files handled. This mode is often less time-consuming and can be used in scenarios to quickly check the result of a part of the generation process.
Deploy	An optional mode that allows for expander handlers to run deployments in isolation. For example, when a developer wants to deploy the output to an Azure App Service.
Migrate	An optional mode that allows for expander handlers to run migrations in isolation. For example, this currently updates the database schema by running the Entity Framework Commandline Interface (see `https://learn.microsoft.com/en-us/ef/core/cli/dotnet`).

TABLE B.2: The available *Generation modes*

```
public class ExpandDatabaseContextHandlerInteractor
    : IExpanderHandlerInteractor<CleanArchitectureExpander>
{
    // ... other code
    public bool CanExecute =>
        parameters
            .GenerationMode
            .HasFlag(GenerationModes.Default) ||
        parameters
            .GenerationMode
            .HasFlag(GenerationModes.Extend);
    // ... other code
}
```

LISTING B.4: Example on how an expander handler can adhere to the RunMode parameters

C Designs

C.1 Legenda

In order to visualize the designs of the artifact, a standard UML notation is used. The designs containing relationships adhere to the following definitions.

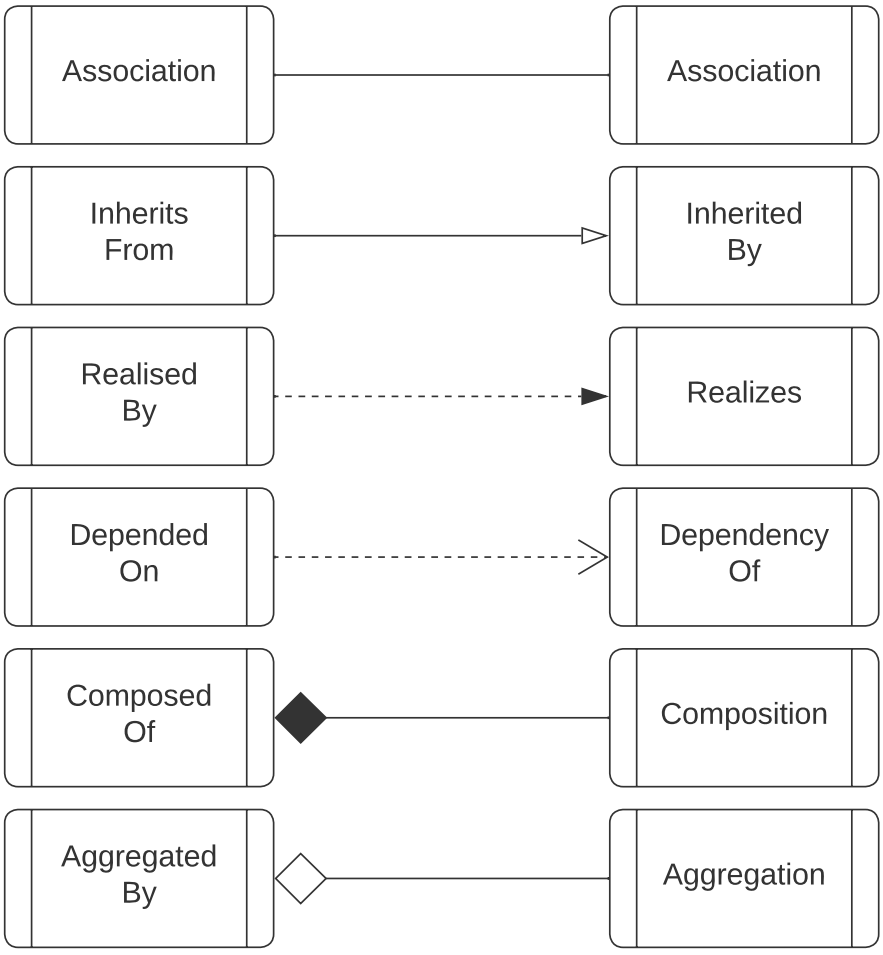


FIGURE C.1: UML notation

C.2 Generic design

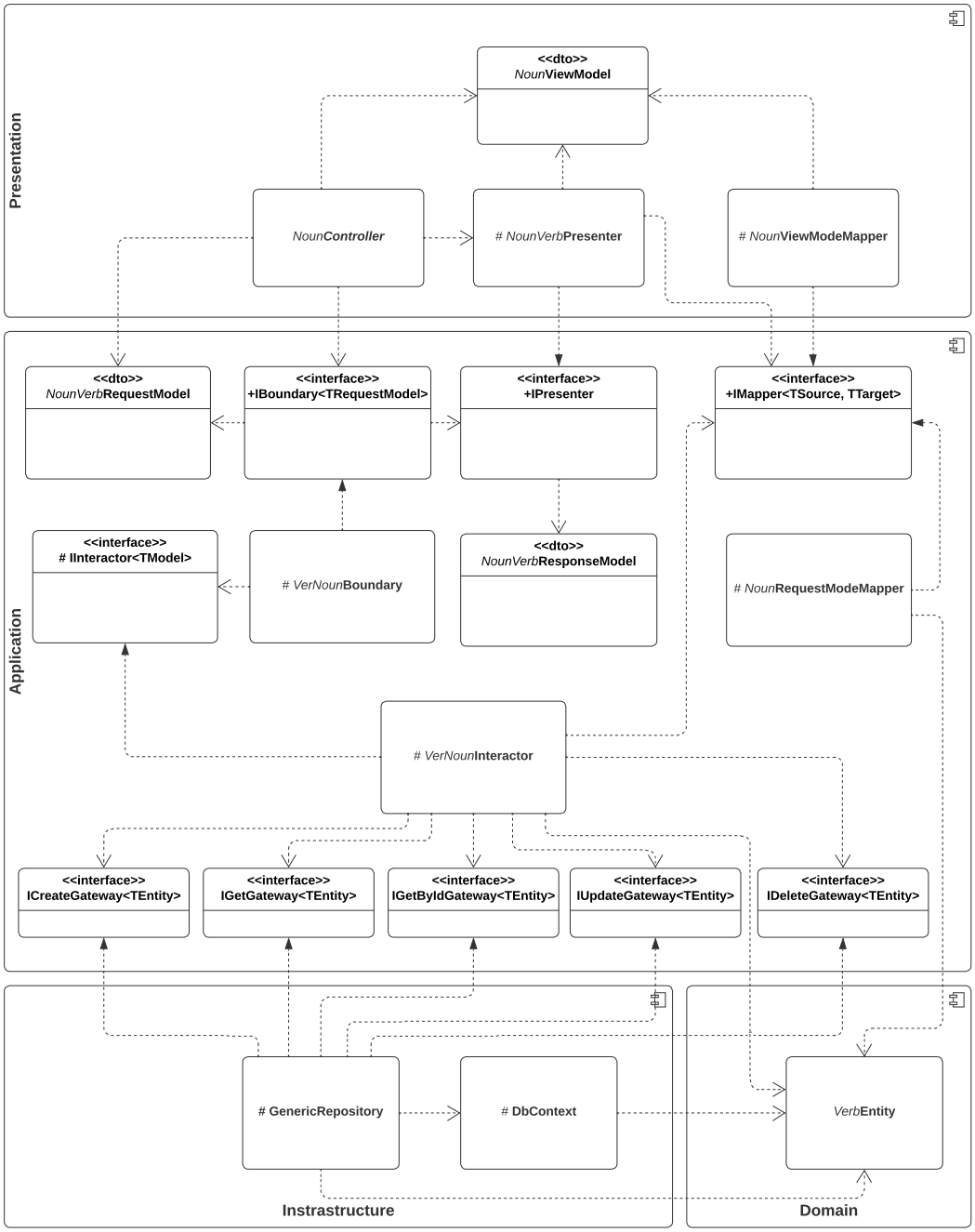


FIGURE C.2: The Generic architecture of the artifacts

Bibliography

De Bruyn, Peter, Herwig Mannaert, Jan Verelst, and Philip Huysmans (Feb. 2018). "Enabling Normalized Systems in Practice – Exploring a Modeling Approach". In: *Business & Information Systems Engineering* 60.1, pp. 55–67. ISSN: 2363-7005, 1867-0202. DOI: 10.1007/s12599-017-0510-4. URL: http://link.springer.com/10.1007/s12599-017-0510-4 (visited on 04/23/2022).

Dijkstra, E (Mar. 1968). "Letters to the Editor: Go to Statement Considered Harmful". In: *Communications of the ACM* 11.3, pp. 147–148. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362929.362947. URL: https://dl.acm.org/doi/10.1145/362929.362947 (visited on 03/20/2023).

– (1982). *Selected Writings on Computing: A Personal Perspective*. Texts and Monographs in Computer Science. New York: Springer-Verlag. 362 pp. ISBN: 978-0-387-90652-2.

Hevner, Alan R, Salvatore T March, Jinsoo Park, and Sudha Ram (n.d.). "Design Science in Information Systems Research". In: ().

Lehman, M.M. (1980). "Programs, Life Cycles, and Laws of Software Evolution". In: *Proceedings of the IEEE* 68.9, pp. 1060–1076. ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805. URL: http://ieeexplore.ieee.org/document/1456074/ (visited on 04/25/2022).

Mannaert, Herwig and Jan Verelst (2009). *Normalized Systems Re-Creating Information Technology Based on Laws for Software Evolvability*. Kermt: Koppa. ISBN: 978-90-77160-00-8.

Mannaert, Herwig, Jan Verelst, and Peter De Bruyn (2016). *Normalized Systems Theory: From Foundations for Evolvable Software toward a General Theory for Evolvable Design*. Kermt: nsi-Press powered bei Koppa. 507 pp. ISBN: 978-90-77160-09-1.

Meyer, Bertrand (1997). *Object-Oriented Software Construction*. 2nd ed. Upper Saddle River, N.J: Prentice Hall PTR. 1254 pp. ISBN: 978-0-13-629155-8.

Oorts, Gilles, Philip Huysmans, Peter De Bruyn, Herwig Mannaert, Jan Verelst, and Arco Oost (Jan. 2014). "Building Evolvable Software Using Normalized Systems Theory: A Case Study". In: *2014 47th Hawaii International Conference on System Sciences*. 2014 47th Hawaii International Conference on System Sciences (HICSS). Waikoloa, HI: IEEE, pp. 4760–4769. ISBN: 978-1-4799-2504-9. DOI: 10.1109/HICSS.2014.585. URL: http://ieeexplore.ieee.org/document/6759187/ (visited on 07/05/2022).

P. Naur and B. Randell (1968). "NATO SOFTWARE ENGINEERING CONFERENCE 1968". In.

Parnas, DL. (Dec. 1972). "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Communications of the ACM* 15.12, pp. 1053–1058. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/361598.361623. URL: https://dl.acm.org/doi/10.1145/361598.361623 (visited on 03/19/2023).

Robert C. Martin (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. London, England: Prentice Hall. 404 pp. ISBN: 978-0-13-449416-6.

Wieringa, Roel J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-662-43838-1 978-3-662-43839-8. DOI: 10.1007/978-3-662-43839-8. URL: http://link.springer.com/10.1007/978-3-662-43839-8 (visited on 06/27/2022).

Code Examples

Koks, GC (2023a). *AbstractExpander*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Expanders/AbstractExpander.cs.

– (2023b). *AbstractExpanderDependencyManagerInteractor*. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Expanders/AbstractExpanderDependencyManagerInteractor.cs.

– (2023c). *AppSeederInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Application/Interactors/Seeders/AppSeederInteractor.cs.

– (2023d). *CleanArchitectureExpander*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/tree/master-thesis-artifact/Expanders/src/PanthaRhei.Expanders.CleanArchitecture.

– (2023e). *CodeGeneratorInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Application/Interactors/Generators/CodeGeneratorInteractor.cs.

– (2023f). *ComponentViewModel*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Presentation.Api/ViewModels/ComponentViewModel.cs.

– (2023g). *DeleteComponentCommand*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/PanthaRhei.Output/Output/6c6984a1-c87a-429b-b91f-2a976adb3c0e/LiquidVisions.PanthaRhei.Generated/src/Application/RequestModels/Components/DeleteComponentCommand.cs.

– (2023h). *ExpandEntitiesHandlerInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Expanders/src/PanthaRhei.Expanders.CleanArchitecture/Handlers/Domain/ExpandEntitiesHandlerInteractor.cs.

– (2023i). *GenericRepository*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Infrastructure.EntityFramework/Repositories/GenericRepository.cs.

– (2023j). *HarvestRepository*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Infrastructure/HarvestRepository.cs.

Koks, GC (2023k). *ICreateGateway*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Gateways/ICreateGateway.cs.
– (2023l). *IExecutionInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/IExecutionInteractor.cs.
– (2023m). *IExpanderHandlerInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/IExpanderHandlerInteractor.cs.
– (2023n). *IExpanderInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Expanders/IExpanderInteractor.cs.
– (2023o). *IHarvesterInteractor*. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Harvesters/IHarvesterInteractor.cs.
– (2023p). *InstallDotNetTemplateInteractor*. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Preprocessors/InstallDotNetTemplateInteractor.cs.
– (2023q). *IRejuvenatorInteractor*. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Rejuvenator/IRejuvenatorInteractor.cs.
– (2023r). *PanthaRheiGeneratorDomain*. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/tree/master-thesis-artifact/Generator/src/PanthaRhei.Generator.Domain.
– (2023s). *PostProcessorInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/PostProcessors/PostProcessorInteractor.cs.
– (2023t). *PreProcessorInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Preprocessors/PreProcessorInteractor.cs.
– (2023u). *RegionHarvesterInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Harvesters/RegionHarvesterInteractor.cs.
– (2023v). *RegionRejuvenatorInteractor*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/Rejuvenator/RegionRejuvenatorInteractor.cs.
– (2023w). *The Component Entity*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Entities/Component.cs.

– (2023x). *The Generator Console*. GitHub. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Presentation.Cli/Program.cs.
– (2023y). *UnInstallDotNetTemplateInteractor*. URL: https://github.com/LiquidVisions/LiquidVisions.PanthaRhei/blob/9687c96eb368d96201d4baa66b1b0536cb90c12c/Generator/src/PanthaRhei.Generator.Domain/Interactors/Generators/PostProcessors/UnInstallDotNetTemplateInteractor.cs.