Concept Alternatives for the Management of Architectural Decisions in Clean Architectures

Raphael Schellander
Supervised by Olaf Zimmermann

Eastern Switzerland University of Applied Sciences

February 23, 2025

Abstract — As software systems become more complex, it is increasingly important to design architectures that can adapt to change, scale effectively, and remain easy to manage. Clean Architecture is a popular approach that helps achieve these goals by organizing systems into layers, each with clear responsibilities. However, applying Clean Architecture can be challenging, especially when it comes to making and keeping track of important architectural decisions. The existing decision management tools often lack domain-specific features and can be difficult to use. There is a need for a better tool that not only helps document decisions, but also guides users through making them in a structured and consistent way. The objective of this project was to design a new tool concept to address the shortcomings of current tools for managing architectural decisions, particularly in the context of Clean Architecture. This involved analyzing existing tools to understand their limitations, developing a conceptual framework for a new tool that guides users through architectural decision-making, and creating an initial proof-of-concept to demonstrate feasibility. The envisioned tool should help users make decisions in a guided manner and to document those decisions in a way that supports the development of the software. The second objective was to analyze and collect key architectural decisions that repeatedly occur when working with Clean Architecture. The resulting collection is envisioned to guide software architects and developers effectively when they make decisions that align with the key principles of Clean Architecture. This collection also serves as an example of a guidance model that shows the viability of the proposed tool concepts. The project resulted in an outline for the new tool and a proof-of-concept implementation in Go. The new tool will have an easy-to-use command line interface. It works in combination with the Clean Architecture decision collection that provides concise step-by-step guidance on key architectural decisions such as setting up the initial structure of the system, defining key components and choosing how different parts of the system will interact. The proof-of-concept, while simple, demonstrates how such a tool could work and lays the foundation for further development. Future enhancements could include features such as interactive decision workflows, integration with software development environments, and advanced analysis tools (for example, machine learning) to help make better decisions. Overall, the tool together with the proof-of-concept provide a suited starting point for improving the way in which architectural decisions are made and documented, helping to create more flexible and maintainable software systems.

Contents

1 Introduction

When developing complex software systems, making informed architectural decisions is fundamental to the success and longevity of the system. These decisions affect certain aspects of the software, from its structure and functionality to its scalability and maintainability. As systems grow in size and complexity, the need for an architectural framework becomes increasingly important. One such framework that has gained widespread attention is Clean Architecture introduced by Robert C. Martin [1], a style of software architecture that emphasizes the separation of concerns and the independence of business logic from external frameworks. However, within the context of Clean Architecture, making informed architectural decisions can be challenging, particularly when trying to balance project-specific needs with the overarching principles of the framework.

This thesis aims to address the knowledge gap in architectural decision-making within the Clean Architecture framework by providing a practical guide tailored to software architects, developers, and technical leads. These professionals, who are responsible for making and managing architectural decisions in complex software systems, are the primary audience of this work. The goal is to provide them with the tools and insights necessary to make decisions that align with Clean Architecture principles. The first part of this thesis includes a thorough exploration of the foundational concepts of Clean Architecture (Section 2) and architectural decisions (Section 3.1). This involves detailing how Clean Architecture principles influence architectural decision-making and how Architectural Decision Records (ADRs) serve as a practical tool for documenting these decisions. Following this theoretical foundation, the thesis includes an analysis of some of the existing tools for managing ADRs in Section 3.2.1. Additionally, a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis for one specific tool is included in Section 3.2.2 to provide a deeper understanding of its applicability. Following the analysis of existing tools is the development of functional and non-functional requirements for a new tool concept aimed at improving the management of architectural decisions in Section 3.2.3. These requirements and user stories are based on the insights gained from the tool analysis and reflect the practical needs of software architects and developers. The resulting concept serves as a blueprint for a proof-of-concept implementation. Furthermore, the thesis explores and documents architectural decisions that are particularly relevant to Clean Architecture in Section 4. This collection should be comprehensive enough to ensure no critical aspects are overlooked, yet concise enough to remain accessible. This practical guide aims to help architects and developers in navigating the complexities of Clean Architecture, enabling them to make informed decisions that are consistent with both the framework's principles and the unique needs of their projects.

Ultimately, this thesis aims to provide a well-rounded understanding of how to effectively manage architectural decisions in systems, especially using Clean Architecture, along with a practical concept for a tool that addresses current limitations in ADR management. The following sections begins with the foundational concepts by looking at the key software principles and architectural layers of Clean Architecture. We then look at what architectural decisions are, why they are important, and how they can be effectively managed using ADRs.

2 Clean Architecture

Clean Architecture is a style of software architecture introduced and popularized by Robert C. Martin in his book "Clean Architecture: A Craftsman's Guide to Software Structure and Design" [1]. This architecture emphasizes the separation of concerns by organizing the codebase into layers, each with different responsibilities. The core idea is to ensure that the business logic, i.e., the most critical and stable part of the system (referred to as policy), remains unaffected by external factors such as frameworks, databases and user interfaces (referred to as details). This independence from external factors allows for another key principle of Clean Architecture, the ability to delay and defer decisions. According to Martin, a good software architect decouples the policy from the details so that decisions about the details can be delayed and deferred [1, p. 140]. For example, the decision of whether to deploy on the web, the cloud or a mobile device should not affect the core business rules. This level of flexibility is critical in environments where technologies and requirements can change rapidly. The origins of Clean Architecture can be traced back to earlier architectural styles, such as the Hexagonal Architecture introduced by Alistair Cockburn [2] and the Onion Architecture proposed by Jeffrey Palermo [3]. Martin built on these ideas to develop a more general approach that could be applied to a wide range of software projects. [1, p. 202] [4]

2.1 Software Design Principles

Clean Architecture is based on software design principles which are important to understand as they build the foundation of any good architecture. This includes the SOLID Principles as well as the Component Principles, and Martin devotes significant attention to these principles in his book.

2.1.1 SOLID Principles

The *SOLID Principles* are a set of five design guidelines that serve as the basis for creating what Martin refers to as clean code. These principles were introduced by Martin and are widely recognized as best practice in software engineering intended to make code more understandable, flexible, and maintainable. Using the analogy of real-world architecture, Martin describes the code as the bricks that are used

to build walls and rooms, and "if the bricks are not well made, the architecture of the building does not matter much." [1, p. 57] The five principles are:

- **Single Responsibility Principle (SRP):** This principle states that a class should have only one reason to change, i.e. it should have only one job (which is not clearly defined and dependent on your specific business logic). By ensuring that each class or module has a single responsibility, the code becomes more modular and easier to understand, test and maintain. [1, p. 61-67]
- **Open/Closed Principle (OCP):** According to this principle, software entities (such as classes, modules and functions) should be open to extension but closed to modification. This means that the behavior of a module can be extended without modifying its source code, protecting existing functionality while allowing new features to be added. [1, p. 69-75]
- **Liskov Substitution Principle (LSP):** This principle states that objects of a superclass should be replaceable by objects of a subclass without affecting the correctness of the program. In other words, a subclass should reinforce, not weaken, the expectations set by its parent class, ensuring that derived classes can stand in for their parent classes without causing errors. [1, p. 77-82]
- **Interface Segregation Principle (ISP):** This principle recommends that clients should not be forced to depend on interfaces that they do not use. This principle encourages the creation of smaller, more specific interfaces rather than large, general-purpose ones, which helps to reduce unnecessary dependencies and increase code modularity. [1, p. 83-86]
- **Dependency Inversion Principle (DIP):** The final principle advises that code which implements high-level policy should not depend on the code that implements low-level details; instead, details should depend on policies. This principle promotes decoupling by ensuring that the high-level code relies on interfaces or abstract classes rather than concrete implementations of low-level details, making the system more flexible and easier to adapt to change. [1, p. 87-91]

2.1.2 Component Principles

The *Component Principles* play a crucial role in the way software modules, or a collection of code, is structured in software systems. These principles are primarily concerned with ensuring that the system remains modular, maintainable, and scalable as it evolves. Using the real-world analogy again, these principles provide guidelines on how the walls should be arranged into rooms and buildings as "large software systems, like large buildings, are built out of smaller components." [1, p. 93] The Component Principles can be divided into two categories; *Component Cohesion* and *Component Coupling*.

Component Cohesion deals with the questions which classes belong in which components and it includes three principles:

- **Reuse/Release Equivalence Principle (REP):** This principle states that the granularity of a component should be based on its potential for reuse and its ability to be independently released. Essentially, a component should be designed so that it can be easily reused and released as a standalone unit. [1, p. 104-105]
- **Common Closure Principle (CCP):** According to this principle, classes that change for the same reason and at the same time should be grouped together into a single component. This ensures that when changes are necessary, they are localized, reducing the impact on the overall system. [1, p. 105-107]
- **Common Reuse Principle (CRP):** This principle advises that classes that are used together should be packaged together. If one class is reused, the others are likely to be reused as well, minimizing the risk of depending on parts of a component that are not needed. [1, p. 107-108]

Component Coupling deals with the relationships between components and also consists of three principles:

- **Acyclic Dependencies Principle (ADP):** This principle dictates that the dependency structure between components should never form cycles. This ensures that the system remains flexible and that components can be independently developed and maintained. [1, p. 112-119]

- **Stable Dependencies Principle (SDP):** According to this principle, a component should only depend on components that are more stable than itself. Stability here refers to the likelihood of change; more stable components are less likely to change, providing a solid foundation for dependent components. [1, p. 120-125]
- **Stable Abstractions Principle (SAP):** This principle complements the SDP by stating that a component should be as abstract as it is stable. Stable components should be highly abstract, meaning that they define interfaces and abstract classes rather than concrete implementations, thus making them more resilient to change. [1, p. 126-132]

Together, the SOLID Principles and the two categories of the Component Principles provide a strong foundation and similar themes can be recognized throughout the layered architecture style of Clean Architecture which is discussed in the following section.

2.2 Architecture Layers

In Clean Architecture, the system is organized into four main layers, as shown in Figure 1, where the outer layers can depend on the inner layers, but not vice versa, which is known as the dependency rule. Figure 1 is a similar representation of the layers to that used in the book, but it includes more information about the contents of the two inner layers (as this is missing from the original diagram) and the titles of the layers are placed inside the rings with an underscore for more clarity. This section explains each layer, starting from the inner layer, the most important, to the outer layer, the least important. For better illustration, we will use an example and apply it to each layer, which is also shown in a similar diagram in Figure 2. For this example, consider the development of an e-commerce shop, that allows registered customers to browse a catalogue of products and place orders online, using an external system such as PayPal or Stripe for payment.

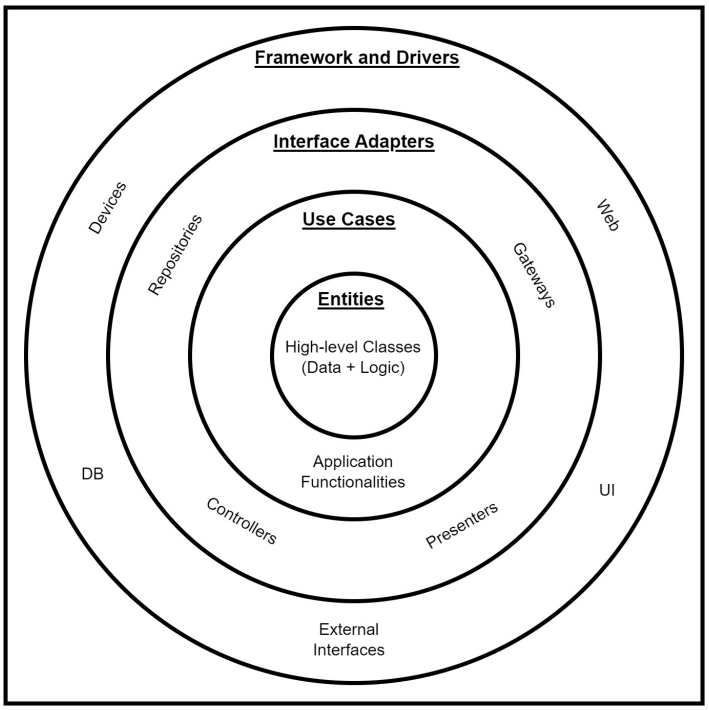

Figure 1: Clean Architecture layers

2.2.1 Entities

Entities represent the most general and high-level rules of the application. They are central to the domain and, in the case of an enterprise, encapsulate the enterprise-wide business rules and can therefore be used by many different applications in the enterprise. An entity can be an object with methods or a set of data structures and functions. They are the least likely to change and therefore have no dependencies on any other layer. [1, p. 204]

For example, an e-commerce shop as described in the beginning would include entities like 'Payment', 'Product', 'Customer', as shown in Figure 2, and many more that are critical to the business model of an e-shop. A 'Payment' entity would include information such as the payment ID, amount, currency, and status. It might also include simple logic to calculate the total amount after applying a discount, determine if the payment is complete based on its status, or verify if the payment amount is within valid ranges (e.g., minimum order amount). A 'Product' entity would contain attributes like product ID, name, description, price,

and stock level, with logic to adjust stock levels after a purchase, and a check if the product is in stock. Similarly, a 'Customer' entity would include attributes such as customer ID, name, email, and address, which might also include logic to validate the format of the customer's email, check if the customer is eligible for a loyalty discount, or update the customer's address when a new one is provided.

2.2.2 Use Cases

While entities can be thought of as enterprise-wide business rules, *Use Cases* are application-specific business rules. Use cases are specific functionalities or features of the application that do not change unless the operational requirements of the application change. They orchestrate the flow of data to and from the entities and use their functionality to achieve a specific goal. Changes to the entities can affect this layer, but not vice versa, and changes to any of the outer layers do not affect the use cases according to the dependency rule. [1, p. 204] However, some use cases may require functionalities from external systems and this is where the dependency inversion principles as described in Section 2.1.1 comes into play. By using abstractions instead of implementations, the use case never creates any dependencies to an outer layer, and only during runtime will the abstraction get replaced with the implementation, for example by using a pattern called dependency injection [5].

Consider now the example use cases shown in Figure 2. The 'ProcessPayment' use case would be responsible for handling the entire payment workflow. This use case would retrieve the total cost of the order from the 'Product' entity, create a 'Payment' entity with the relevant details, and then interact with an external payment service (using an abstraction) to process the payment. If the payment is successful, the 'Payment' entity would be updated accordingly, and control could be passed to another use case, such as 'UpdateInventory', which would manage the stock levels of products after a purchase. It would retrieve the necessary product information from the 'Product' entity and decrease the stock levels based on the quantity purchased. This use case ensures that inventory is accurately tracked, preventing overselling and maintaining up-to-date stock information. The 'RegisterCustomer' use case, on the other hand, would handle the process of registering a new customer. It would gather customer details, such as name, email, and address, validate the data (perhaps by using some basic logic within the 'Customer' entity), and then store the new customer information in the database by interacting with the 'Customer' entity.

As you can see, one use case does not necessarily only make use of just one entity. For example, 'ProcessPayment' would need information from the 'Product' entity to calculate the accurate amount for all products to create a 'Payment' entity with the correct total. Also, use cases can be used together in succession; for instance, after the payment is processed and verified by 'ProcessPayment', the 'UpdateInventory' use case would ensure that the stock of the purchased items is updated accordingly.

2.2.3 Interface Adapters

Interface Adapters serve as a bridge that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external systems, such as the user interface or the database. [1, p. 205] Within this layer, several key patterns are commonly used to manage different aspects of the system's interaction with external components.

Gateways [6] are used to abstract interactions with external systems such as databases, or third-party APIs. They act as an interface between the internal use cases and these external systems, ensuring that the core business logic does not directly depend on these external systems. For instance, the 'PaymentGatewayAdapter' in our e-commerce platform would handle communication with payment services like PayPal or Stripe by handling the protocols and translating data formats from our application, like the 'Payment' entity, to a format required by these services.

Repositories [7] are responsible for managing the communication between the application and the data source, typically a database but it could also be the file system or cloud storage. They abstract the complexities of data storage and retrieval, ensuring that the use cases do not need to interact directly with the database. This separation allows for easier maintenance and the flexibility to change the data source or the way data is stored without impacting the business logic. For Example 'InventoryRepository' is used to interact with and update the database when the 'UpdateInventory' use case has changed the stock amount of a 'Product'.

Controllers [8] are responsible for handling user input and directing it to the appropriate use case, interpreting requests from the user interface and invoking the corresponding use case with the necessary data. In our e-commerce example, an 'CustomerController' would handle HTTP requests for registering a customer,

extract the relevant data, and pass it to the 'RegisterCustomer' use case for processing.

The presenter pattern [9] is employed to manage UI logic by transforming the data output from the use cases into a format suitable for display by the user interface. This ensures that the UI layer remains as simple as possible, with all the data formatting and presentation handled by the presenter. In our e-commerce example, we have a 'CustomerPresenter' for showing the customer information in the profile tab, transforming the data from the 'Customer' entity into a format that is easily rendered by the front-end framework. This might involve formatting dates, combining first and last names into a full name, or organizing data into a layout that fits the design of the profile page.

2.2.4 Frameworks and Drivers

Frameworks and Drivers is the outermost layer of the architecture, encompassing the various frameworks, tools, and services upon which the system depends, including devices, web frameworks, user interfaces, databases, and external interfaces. This layer is considered the most volatile, meaning it is most susceptible to change as technologies evolve and new requirements emerge. One of the key principles of Clean Architecture is to keep the core business logic isolated from these external technologies, allowing them to be replaced or upgraded without affecting the inner layers. [1, p. 205-206]

For example, in our e-commerce platform as shown in Figure 2, a web framework like React or Angular would handle the requests coming from a browser. This could be a request from a customer to create a new account, which would be passed to the 'CustomerController', then to the 'RegisterCustomer' use case, and when the operation is complete would notify the 'CustomerPresenter', which would finally prepare the data for the browser to show the customer their profile page after successful registration. This is a very simple example of the entire workflow, starting from the outermost layer going inwards to the use case and entities layer, before coming out again, and in a real project such a registration process would probably involve many more controllers, use cases, entities, etc. The database used by our e-commerce platform is another example of the benefit of isolating this layer. If we initially use a SQL database, but later need to switch to a NoSQL solution to handle more complex queries or larger datasets, this change would only affect the frameworks and drivers layer. Use cases and abstractions that make changes to the database store, such as the 'UpdateInventory' use case

and the 'InventoryRepository', have no dependencies on the actual framework used. Finally, we can see a similar benefit for the way our e-commerce shop handles payments. We have the benefit of implementing multiple options for the user, as the 'ProcessPayment' use case works the same no matter what system we use, for example PayPal or Stripe, and the 'PaymentGatewayAdapter' will choose accordingly.

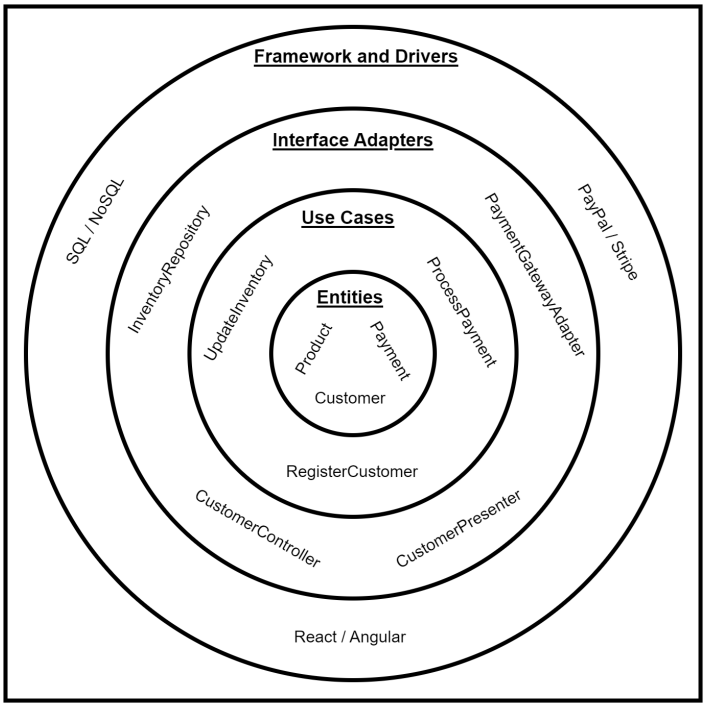

Figure 2: Clean Architecture layers; e-commerce shop example

The goal of Clean Architecture is to create systems that are resilient to change, easy to understand and easy to maintain. Similar to the Single Responsibility Principle we have seen in Section 2.1.1, Clean Architecture allows for a clear separation of concerns by organizing the code base into distinct layers, each with specific responsibilities. This separation ensures that changes to one part of the system do not inadvertently affect other parts, making it easier to develop, maintain and test each layer independently, while reducing the risk of introducing errors into other layers when making changes. The Dependency Rule, which says that source code dependencies must point only inward, allows for another key benefit of Clean Architecture, the ability to defer decisions. The ability to defer certain decisions, such as the choice of database technology or UI framework, allows the development team to focus on the core business logic first.

3 Architectural Decisions

Architectural decisions play an important role in influencing the non-functional characteristics or qualities of a software system, such as performance, flexibility, scalability and maintainability. Each decision addresses a specific, architecturally significant design problem for which there are several possible solutions. The decision process involves selecting the most appropriate solution by carefully considering how it will affect the desired quality attributes of the system. These decisions may apply to the entire software system, or may focus on only one or more of its core components. [10] As Jansen and Bosch highlight, making architectural design decisions explicit is essential to prevent knowledge from being lost, which can lead to increased complexity and higher maintenance costs. By focusing on these decisions, we can create a system architecture that is more robust, easier to maintain, and adaptable to new requirements. [11] Documenting these decisions is important to maintain transparency and ensure that developers and stakeholders understand the reasoning behind them. It is also useful for reviewing and remembering why a decision was made, along with its outcome. This is where Architectural Decision Records (ADRs) come in. ADRs are a structured way of capturing the context, rationale and consequences of architectural decisions. [12] They are an essential tool for tracking the evolution of the system's architecture over time. ADRs are discussed in more detail in Section 3.1.

To better understand the architectural decision-making process, consider the process flow diagram shown in Figure 3. This diagram outlines an example of a structured and iterative approach to making and documenting architectural decisions. It is important to note that this is not a fixed or universally defined process, but rather a simple framework for illustrative purposes, which can be adapted to suit the specific needs of your project. The process begins with the identification of an architectural problem or requirement. Once identified, the next step is to gather and refine the relevant requirements and constraints, ensuring that all critical factors are considered. This is followed by the evaluation of options, where different architectural solutions are assessed based on how well they meet the analyzed requirements. Once a decision is made, it is documented in an ADR to ensure transparency and traceability. The proposed ADR is reviewed by the relevant stakeholders and either marked as accepted and implemented or as unacceptable, while the process returns to refine the requirements. As the project evolves, the decision can be revisited even after it has been implemented, if any of the requirements have changed, or simply as a periodic update.

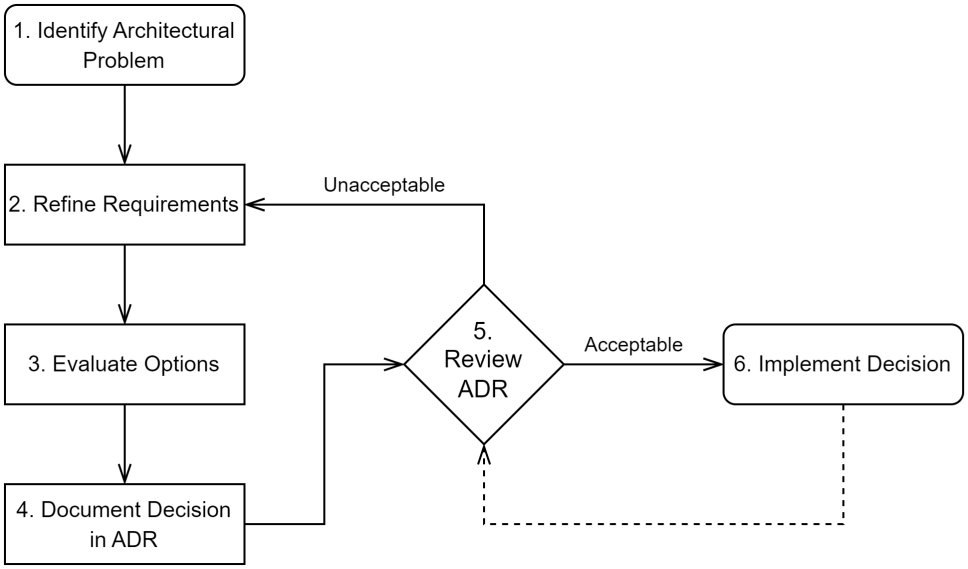

Figure 3: Architectural decision-making process (own presentment)

Consider, for example, the need to determine the architectural style for a new software project:

1. The identified problem is that the project requires a well-defined architectural style that will guide the structure and development of the application. The decision needs to focus on selecting an architecture that fulfills the specific requirements of the project.

2. To make an informed decision, it is essential to gather and refine the project's specific requirements. Key considerations include the necessity for scalability to manage growing user demand, ease of deployment and updates, fault isolation to prevent failures in one part of the system from affecting the whole, and the operational complexity of managing the system. Other constraints, such as the development team's expertise, the time-to-market, and the available budget, must also be taken into account.

3. Two architectural styles are evaluated (for the simplicity of this example) based on how well they meet the requirements of the project. The first option is a monolithic architecture style; this approach builds the application as a single, cohesive unit, and while this simplifies initial development and deployment, it can make it difficult to scale the system as the application grows. The second option

is a microservices architecture style; this style divides the application into smaller, independently deployable services and offers benefits such as independent scaling of services, better fault isolation and the ability to deploy updates to individual services without affecting the entire system, but it also introduces complexity in areas such as inter-service communication, data consistency and management of distributed transactions.

4. After carefully evaluating the options, a decision is made based on how well each architectural style fits the refined requirements. If independent scaling and fault isolation are critical, the team may opt for a microservices architecture. Conversely, if simplicity and rapid initial development are priorities, a monolithic architecture may be the preferred choice. This decision is documented in an ADR, which captures the context, rationale and expected consequences of the chosen architecture. In this case, the decision is made to adopt a monolithic architecture for the project. This choice is based on the need for simplicity and rapid initial development, which are critical in the early stages of the project.

5. The ADR is then reviewed by key stakeholders, including project managers, lead architects and development teams. If the decision is approved, it is implemented as described. If the decision is not approved by the stakeholders, the requirements may be reviewed and the evaluation process repeated to consider other options.

6. If the decision to use a monolithic architecture is accepted, the system is developed accordingly. As the project evolves, particularly as it increases in size and complexity, the architectural decision may be revisited. For example, as the application scales and new features are added, the development team might consider moving to a microservices architecture to handle the increased complexity and improve scalability. This potential shift would also be documented and reviewed based on the new requirements to ensure that the architecture continues to effectively meet the needs of the project.

3.1 Architectural Decision Records

Architectural Decision Records (ADRs) are a structured approach to documenting the architecturally significant decisions made during the design and development of a software system. The practice of documenting these decisions ensures that the reasoning behind each choice is clearly understood, not only at the time the decision is made, but also in the future when the system may be modified, or simply as a reminder to avoid repeated discussions and having to make the decision again

because it has been forgotten. The concept of ADRs was popularized by Michael Nygard in a blog post [12], where he introduced the idea of capturing architectural decisions in a lightweight and accessible format. Nygard argues that traditional documentation methods often fail to stay current, leading to a disconnect between the intended architecture and its actual implementation. He advocates for ADRs as a more lightweight and effective approach to documenting architectural decisions. ADRs allow teams to capture the reasoning behind decisions in a way that is easier to maintain and update, ensuring that the architecture remains coherent and aligned with the project's development. It is essential to integrate ADRs into the team's development workflow, making them part of the decision-making process rather than an afterthought and several tools exists to facilitate this process. We will have a look at some of these available tools in Section 3.2.

An ADR should be clear, concise and focused on a single decision, making it easier for future teams to understand and, if necessary, revise the decision based on new information or changing circumstances. To maximize the effectiveness of ADRs, it is important to maintain consistency across the project by using a standardized format for all records. This consistency makes it easier to navigate and understand the documented decisions over time. In addition to the format suggested in Nygard's original blog post, several others have been created to standardize the documentation of ADRs. These formats vary in complexity and focus, allowing teams to choose the one that best suits their project's needs. Figure 4 shows three different formats of recording architectural significant decisions, which will be discussed in more detail in the following sections. The formats includes the one originally outlined by Nygard, YStatements, and Markdown Architectural Decision Records (MADR). It is important to note that whichever format is chosen, it can be adapted to suit specific needs.

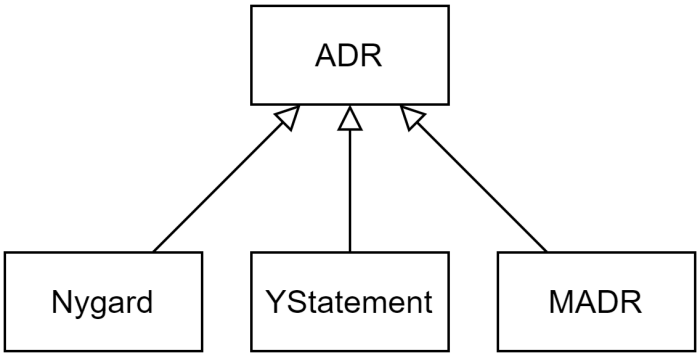

Figure 4: Formats for documenting Architectural Decision Records (ADRs) [13]

3.1.1 Nygard Format

The *Nygard format*, introduced by Michael Nygard in his influential blog post [12], is one of the most widely recognized and adopted approaches to documenting Architectural Decision Records (ADRs). The format is designed to be lightweight and straightforward, focusing on capturing the essential aspects of an architectural decision without overwhelming the reader with unnecessary details. The simplicity of the format makes it accessible and easy to integrate into the development process, ensuring that it is used consistently and effectively across the project.

The Nygard format typically includes the following components:

- **Title:** The title is a brief and descriptive summary of the architectural decision. It should be concise yet informative, providing enough context for someone skimming through the ADRs to understand the essence of the decision. The title serves as the first point of reference, making it easy to identify the decision at a glance.
- **Context:** This section provides the background and rationale for the decision. It describes the circumstances that led to the need for an architectural decision, including any relevant project goals, constraints, and challenges. The context helps to frame the decision, ensuring that future readers understand the problem that was being addressed. This section is crucial because it situates the decision within the broader scope of the project, allowing others to see why this particular decision was necessary and how it fits into the overall architecture.
- **Decision:** The decision section clearly states what choice was made and why. It should be a definitive statement that leaves no ambiguity about what was decided. This section often includes a brief explanation of why this option was chosen over others, emphasizing the trade-offs that were considered. The decision should be articulated in a way that it can stand alone, providing a clear directive to anyone who needs to implement or understand the outcome.
- **Status:** The status indicates the current state of the decision, such as 'proposed', 'accepted', 'superseded', or 'deprecated'. This allows the team to track the lifecycle of the decision over time, particularly if the decision is revisited or revised as the project evolves. The status helps maintain clarity about whether the decision is currently in effect or if it has been replaced by a new one.

- **Consequences:** This section outlines the expected outcomes of the decision, both positive and negative. It is important to be thorough in documenting the consequences to provide a balanced view of the decision's impact. By documenting the consequences, the ADR provides a foresight into the implications of the decision, helping teams prepare for and mitigate any potential challenges.

The strength of the Nygard format lies in its simplicity. By concentrating on the most critical aspects of a decision — what was decided, why it was necessary, and what the outcomes might be — this format ensures that ADRs are easy to create, maintain, and understand. However, a drawback of the format is that the context section is too broad and does not specify what and how much should be mentioned. This leads some users to provide far too much unnecessary context, bloating the ADR, while others provide too little context, making the subsequent decision outcome unclear. It is also not clear why only a status was included to provide metadata, and not other important information such as a timestamp for the date of creation.

Figure 5 shows an example of how the Nygard format might look like in practice. This could be such an ADR that was documented as part of the decision-making process example in Section 3, where after initially using a monolithic approach, the decision was made to change to microservices due to changing requirements.

Title:

Microservices Architecture for E-Commerce System

Context

We initially chose to implement a monolithic architecture for easy development in the beginning stages of development. Due to increasing traffic and frequent updates, the choice between a microservices and a monolithic architecture has gotten relevant again.

Decision

We will adopt a microservices architecture, dividing the application into smaller, independently deployable services to allow independent scaling, fault isolation, and adaptability to changing requirements.

Status

Accepted (whereas the initial ADR would now be marked as superseded)

Consequences

- Positive: Enhanced scalability, fault isolation, and flexibility in updating and deploying individual services.
- Negative: Increased complexity in managing inter-service communication, data consistency, and distributed transactions.

Figure 5: Architectural Decision Record in Nygard format for using microservices.

3.1.2 Y-Statement

The *Y-Statement format* is a structured approach to documenting architectural decisions in a single, comprehensive sentence by focusing on the rationale behind them, i.e., to clarify why ('Y') a decision was made, and was introduced by Olaf Zimmermann in a presentation at the SATURN 2012 conference [14] and elaborated upon in various publications and articles. [15] [16] It addresses six critical aspects that encapsulate the essence of the decision-making process, starting with outlining the situation (context, requirements, options) before narrowing down to a specific decision and providing the reasoning (results, consequences). More specifically, the structure of a Y-statement is as follows:

- ***In the context of <use case and/or component>:*** This part sets the stage by describing the situation or environment in which the decision is being made. It outlines the specific circumstances that are driving the need for a decision.
- ***Facing <non-functional concern>:*** This section identifies the key requirements or challenges that the decision must address. These could be technical constraints, business needs, performance goals, or any other factors that are influencing the decision.
- ***We decided <option 1>:*** Here, the decision itself is clearly stated. This is the choice that has been made in response to the context and requirements.
- ***And neglected <option 2..n>:*** This part acknowledges the alternatives that were considered but ultimately not chosen. Including this information provides insight into the decision-making process and shows that different options were assessed.
- ***To achieve <quality>:*** This section explains the intended result or benefit of the decision. It describes what the decision is expected to accomplish.
- ***Accepting that <consequences>:*** Finally, this part documents the trade-offs or potential negative consequences of the decision. This ensures that the downsides of the decision are understood and accepted by the team.

Y-Statements improve on the criticism of the Nygard format by adding a non-functional requirements section to provide more specific context, including the neglected options next to the decision, and mentioning the consequences of the decision next to the expected result. However, it does not include any metadata such as a status or timestamp.

Figure 6 shows an example of how a Y-statement could be used to describe another architectural decision in our e-commerce shop. The Y-Statement captures the entire decision-making process in a concise, easy-to-understand manner.

```
In the context of building an e-commerce platform with a need for
fast and reliable order processing,

facing the requirement to minimize downtime during peak shopping
seasons,

we decided to implement a queue-based order processing system,

and neglected a real-time processing approach,

to achieve greater reliability and fault tolerance during high
traffic periods,

accepting that this will introduce slight delays in order
confirmation and add complexity to the system.
```

Figure 6: Architectural Decision Record in Y-Statement format for choosing order processing strategy.

3.1.3 Markdown Architectural Decision Records (MADR)

Markdown Architectural Decision Records (MADR) [17] is a format for documenting architectural decisions by leveraging the simplicity of Markdown syntax and was introduced in a puplication by Oliver Kopp, Anita Armbruster, and Olaf Zimmermann [18]. This format is particularly well-suited for integration with version control systems, allowing teams to track changes to their architectural decisions alongside their source code. By using Markdown, MADR makes it easy for developers to create, edit, and review ADRs using familiar tools, while maintaining a consistent and organized structure. MADR provides templates for both a short and a long version of the decision record, giving teams the flexibility to choose the level of detail that best fits their needs.

The short version of MADR is designed for decisions that can be documented in a brief manner, with just enough information to capture the essential details. It includes the following sections [19]:

- **Context and Problem Statement:** This section provides the background and motivation for the decision. It outlines the specific problem or challenge that needs to be addressed.

- **Considered Options:** Here, the various options or alternatives that were considered during the decision-making process are listed. This provides a record of what was evaluated before making the final decision.
- **Decision Outcome:** This section clearly states the decision that was made, summarizing the choice and its justification.

The long version of MADR builds on the short version by adding additional sections that provide a more comprehensive overview of the decision. This version is ideal for more complex decisions that require thorough documentation. It includes [19]:

- **Consequences:** This section outlines the expected results of the decision, both positive and negative. It is important to document these consequences to ensure that the team is aware of the trade-offs involved.
- **Pros and Cons of the Options:** In this section, the advantages and disadvantages of each considered option are listed. This helps to clarify why certain options were chosen over others, providing transparency and insight into the decision-making process.
- **More Information:** This optional section can include any additional details, references, or documentation that are relevant to the decision. It serves as a catch-all for information that does not fit neatly into the other sections but is still important to record.

The complete MADR template is available for download from GitHub [20] and can then be imported into your project repository, where it can be adapted to suit your specific needs. Again, MADR provides more specific context and information about the ADR than the Nygard format, with many optional sections that can be used or not as needed. It also includes more metadata information at the beginning of the file, such as status, date and decision makers. [21]

Figure 7 shows an example MADR using the short version template for a decision related to our e-commerce platform.

```
--
status:  Accepted
date:  2024-09-06
deciders:  RS
--

# Choice of Database system for E-Commerce Platform

## Context and Problem Statement
We are developing an e-commerce platform that needs to support a
growing number of users and transactions.  As the platform expands,
the scalability and performance of our database become critical
concerns.  We need to decide on a database technology that can
handle large volumes of data, ensure quick transaction processing,
and provide flexibility for future growth.

## Considered Options
SQL Database (e.g., PostgreSQL)
NoSQL Database (e.g., MongoDB)
NewSQL Database (e.g., CockroachDB)

## Decision Outcome
Chosen option:  "NoSQL database", specifically "MongoDB", based
on the need for horizontal scalability and flexibility in handling
unstructured data.
```

Figure 7: Markdown Architectural Decision Record format for choosing a database system.

3.2 Tools

As the complexity of software systems increases, so does the importance of efficiently managing all the decisions that have been made. To aid this process, several tools have been developed to assist architects and developers in creating, managing, and visualizing Architectural Decision Records (ADRs). Most of these tools were identified through the ADR GitHub organization website [13], which serves as a central hub for ADR-related resources and tools. Additionally, more general tools not specifically related to ADR were discovered through targeted online searches using terms like 'architectural', 'decision', 'management', and other

related keywords. These searches yielded tools that, while not exclusively designed for ADR management, offer relevant functionality that can be adapted for this purpose. These tools vary in functionality from simple file management utilities to more comprehensive platforms that integrate directly into development environments. The use of ADR tools is motivated by the need for consistency, traceability, and efficiency in the management of architectural decisions. As projects evolve, maintaining an organized and accessible record of ADRs becomes essential. Based on their functionality, the analyzed ADR tools can be broadly categorized into the groups described in the following section.

3.2.1 Tool Comparison

Tools for Visualising and Navigating ADRs: These tools provide interfaces for browsing and visualizing ADRs, often generating logs, web pages or other navigable formats that make it easier to explore existing decision records. They are particularly useful for teams that need to review and reference ADRs on a regular basis.

Tools for Creating and Managing ADRs: These tools focus on the creation, organization and storage of ADRs. These tools often provide features such as templates, integration with version control systems and the ability to embed ADRs directly into the code base, ensuring that documentation is closely linked to the development process. Within this category there are tools specifically designed to allow ADRs to be embedded directly into code as annotations or attributes, making the decision documentation process more natural for developers. There are also command line interface (CLI) tools and integrated development environment (IDE) plug-ins, as well as web-based applications, that simplify the workflow by providing developers with utilities to create and manage ADRs directly from their development environments or web browser.

General Decision-Making Tools: Although not specifically designed for ADRs, these tools support decision-making processes and may be adapted for use in managing architectural decisions. They include features for collaborative decision-making, trade-off analysis and prioritization (these tools are commercial solutions and have not been tested specifically for ADRs).

Table 1 compares twelve ADR tools, categorizes them into one of the three groups mentioned above and describes their primary functionality. In Section 3.2.2 we will take a closer look at one particular tool for further analysis.

Tool Name	Category	Platform	Primary Functionality
adr-log [22]	Visualizing and Navigating ADRs	Command-line Interface (CLI)	Generates a log of ADRs from a specified folder, providing an overview and easy navigation capabilities.
adr-viewer [23]	Visualizing and Navigating ADRs	Web-based	Generates web pages from ADRs, making them easy to read and navigate.
e-adr [24]	Creating and Managing ADRs	Embedded (Java)	Embeds ADRs directly in Java code using annotations.
architectural-decision [25]	Creating and Managing ADRs	Embedded (PHP8)	PHP8 library for documenting ADRs using attributes directly in the codebase.
adr-tools [26]	Creating and Managing ADRs	CLI	Lightweight and easy to use CLI tool for creating and managing ADRs. Provides utilities for working with ADRs directly from the terminal. More details in Section 3.2.2.
talo [27]	Creating and Managing ADRs	CLI	CLI tool for managing, creating, updating, and exporting ADRs and other documents like RFCs.
log4brains [28]	Creating and Managing ADRs	Integrated Development Environment (IDE)	Docs-as-code tool for logging ADRs from an IDE and publishing them as a static website.
pyadr [29]	Creating and Managing ADRs	CLI	CLI tool for managing the ADR process lifecycle.
adr-manager [30]	Creating and Managing ADRs	Web-based	Web application for creating and managing ADRs in Markdown (MADR). User-friendly web interface for managing ADRs.
Loqbooq [31]	General Decision-Making Tool	Web Application, Slack Plugin	Logs and manages team decisions with features like Slack integration, PDF/CSV export, and secure data storage.
Loomio [32]	General Decision-Making Tool	Web Application	Collaborative decision-making platform with tools for consensus-building, voting, and communication.
Expert Choice [33]	General Decision-Making Tool	Application	Decision-making software for prioritizing options, allocating resources, and scenario planning.

Table 1: Comparison of Architectural Decision Record (ADR) tools

3.2.2 Tool Showcase: adr-tools

In this section we will take a closer look at one particular tool, Nat Pryce's *adr-tools* [26], analyzing its features, strengths, weaknesses and potential applications. This analysis will provide insights into how this tool can be used to support and improve the management of architectural decisions. The findings from this analysis will form the basis for an outline of an alternative tool, which will be presented in Section 3.2.3, together with a proof-of-concept implementation. The decision to focus on *adr-tools* for the analysis in this section was based on its focus on the fundamental concepts of managing Architectural Decision Records (ADRs), as well as being a very lightweight tool that is easy to test and analyze. *adr-tools* is one of the most popular tools for ADRs, based on the number of stars it has on GitHub. It is also the basis for several other tools, such as *adr-log* and *adr-viewer*, which are included in the comparison in Table 1. To do a comprehensive analysis of *adr-tools*, we will first provide a simple demonstration of how to use the tool, before continuing with a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis (see Figure 9).

Command	**Description**
adr init <dir>	Initializes a new log of ADRs in the directory <dir>.
adr new <title>	Creates a new, numbered ADR with the given <title>. The number increases with each decision and starts with 0001-<title>. The created ADR uses the Nygard format (see Section 3.1.1) and includes the current date.
adr link <source> <src desc> <trgt> <trgt desc>	Links two ADRs, establishing a relationship between <source> and <target>, represented by their ADR number (e.g. 0001, or simply 1). The description is used in the respective ADR under the status section. Linking the same ADRs multiple times will stack instead of removing any old links.
adr list	Lists all ADRs located inside the directory initialized with 'adr init'.
adr config	Displays or sets configuration options for adr-tools.
adr generate toc	Generates a table of contents for the ADR log.
adr generate graph	Generates a graph of ADRs showing links between them (requires Graphviz installed).

Table 2: Usage of Nat Pryce's *adr-tools* [26]

Usage

To install and set up *adr-tools*, you can either clone the repository or download the latest release from the project's GitHub page. Once you have the files, you should add the 'src' directory to your system's 'PATH' environment variable so that the commands can be used globally from any directory. Depending on your operating system, you can also use package managers such as Homebrew on MacOS or Linux to install *adr-tools* more conveniently. On a Windows system, *adr-tools* can only be used within the Windows Subsystem for Linux (WSL), where you would follow the same setup steps as in a Linux environment. Once these steps have been completed, the tool is ready to use.

As an example we will demonstrate the usage of *adr-tools* commands (as shown in Table 2) by documenting the decision of choosing between a monolithic or a microservices approach as an architectural style:

1. Initialize ADR Directory. Begin by setting up your ADR directory within your project using the 'adr init' command (this only needs to be done the first time). This will create a structured directory for all your ADRs, including an initial ADR with number 1:

```
> adr init doc/adr
```

2. Create a new ADR. Following the example of a decision-making process described in Section 3, we create a new ADR to document the decision for using a monolithic approach for our e-commerce system (ADR 2), as well as another ADR for when we ultimately decided to replace it with a microservices style (ADR 3). This command generates a new markdown file within your 'doc/adr' directory with a template where you can record the context, decision, and consequence (the file will be directly opened in your editor of choice, if configured):

```
> adr new "Monolithic Architecture for E-Commerce System"
> adr new "Microservices Architecture for E-Commerce System"
```

3. Document the Decision. Fill in the generated ADR files with relevant details, as for example seen in Figure 5, in your editor of choice.

4. Link Related ADRs. If you have other ADRs that are related to each other, you can use the 'adr link' command to link them together. The following command for example, establishes a relationship between the two ADRs we created. Under the status section, ADR 2 will say "superseded by" with a link to the ADR 3 file, and ADR 3 will say "supersedes" followed by a link to the ADR 2 file.

```
> adr link 3 "supersedes" 2 "superseded by"
```

5. List ADRs. To review all documented decisions, you can use the 'adr list' command. This provides a quick overview of all ADRs inside the folder initialized in step 1, making it easier to track and reference them:

```
> adr list
```

6. Generate a graph to visualize ADRs. To visually review all documented decisions, you can use the 'adr generate graph' which generates a visualization of the links between decision records in Graphviz format (example Figure 8):

```
> adr generate graph
```

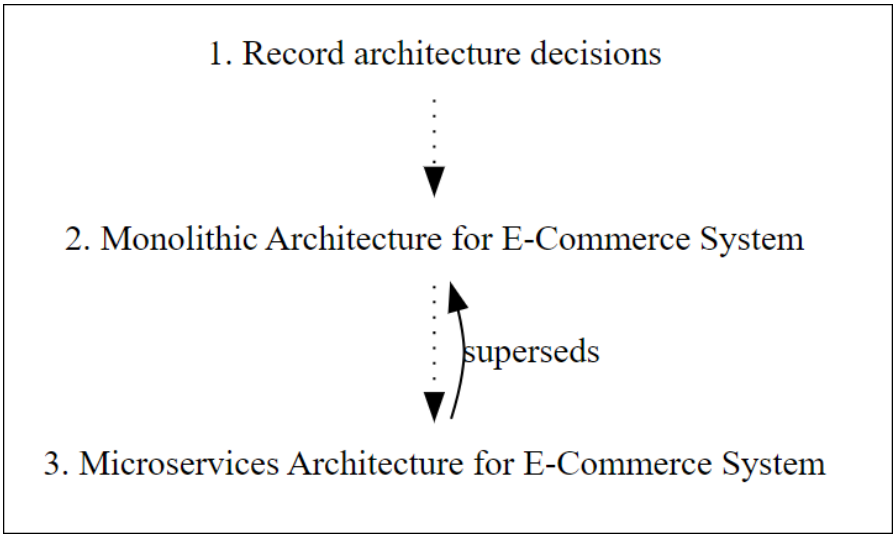

Figure 8: Visual representation of links between ADRs

SWOT Analysis

Strengths: *adr-tools* is characterized by its simplicity and ease of use. The usage through the command-line interface (CLI) allows users to quickly create, view, and manage ADRs with minimal setup or configuration. A key strength of *adr-tools* is its ability to generate consistent and sequentially numbered ADRs, which is particularly valuable for maintaining an organized and traceable decision log. The tool also facilitates linking ADRs, making it easier to navigate through related decisions, a task that would be more cumbersome if done manually in a traditional text editor. This is even further improved by the support to generate a graphical representation of the linked ADRs in Graphviz format. Additionally, ADRs can easily be integrated with Git to ensure that decisions are version-controlled alongside the source code, providing a transparent and historical record of architectural decisions that evolve over time. The open-source nature of *adr-tools* is another strength, as it allows for community contributions, flexibility, and adaptability to different project needs. This adaptability is demonstrated by the fact that *adr-tools* is the basis for several other tools in the ADR domain. Another advantage of *adr-tools* is its long-term compatibility, as it is implemented using shell scripts, a fundamental and stable component of Linux systems.

Weaknesses: While *adr-tools* offers several benefits, it also has notable weaknesses. One of the primary criticisms is that it functions largely as a file creation tool, using a specific template that users must mainly fill in themselves. This raises the question of whether the tool, despite the benefit of linking ADRs, provides enough added value over simply using a text editor to create and manage ADRs. The lack of a Graphical User Interface (GUI) or Integrated Development Environment (IDE) integration could be considered as a drawback, especially for teams that prefer visual tools or have members who are less comfortable with command-line operations. This is also illustrated by the fact that although it is possible to create a graphical representation showing the links between ADRs, *adr-tools* will only create a text output which has to be imported into another tool capable of visualizing the Graphviz format. Furthermore, *adr-tools* does not directly support Windows, limiting its accessibility in development environments that do not utilize POSIX systems like Linux or macOS. Additionally, as *adr-tools* requires manual effort to configure and integrate into a project, it may not offer significant time-saving benefits compared to other tools that provide more automation or built-in integrations.

Opportunities: *adr-tools* has significant potential for growth and innovation within the domain of architectural decision management. While the management of the ADR link is one of the greatest benefits provided, a major opportunity lies in developing the tool into a more comprehensive tool. This could include the integration of commands that allow the user to create and select from templates or models to suit the specific ADR needs of their project, which could greatly increase the value added by using such a tool. Furthermore, features that go beyond simple file creation could be introduced, such as advanced decision analytics and visualization tools that help teams understand the impact of their decisions over time. For instance, *adr-tools* could be enhanced with capabilities to track the interdependencies between ADRs, visualize decision trees, or map out the potential consequences of different architectural choices. Additionally, the tool could integrate with machine learning algorithms to suggest potential decisions based on historical data, industry best practices, or even the current project-specific ADRs. Implementing automation features, such as automated reminders to update ADRs, could further improve usability and ensure that ADRs are maintained consistently.

Threats: *adr-tools* faces several threats that could affect its adoption and relevance. The preference for GUI-based tools as well as IDE plugins, particularly in teams with members that are less familiar with CLI operations, could be a significant threat as these teams may be drawn to tools that offer a more visual and user-friendly experience. In addition, the emergence of new tools with more advanced features could reduce the adoption rate of *adr-tools*. Another threat comes from the growing trend toward automation and AI-driven development processes. As artificial intelligence and machine learning are increasingly used to assist in decision-making and code generation, there is a risk that *adr-tools*, which relies on manual input and management, may be perceived as outdated or insufficiently advanced for modern development practices. If AI-powered tools begin to offer automated ADR generation, suggestions, and real-time impact analysis, developers might prefer these more dynamic solutions over *adr-tools*. If *adr-tools* fails to keep up with these trends, it could see a decline in its user base.

Figure 9: SWOT Analysis of *adr-tools* [26]

3.2.3 Concept Alternative: ADG-Tool

In the previous sections, we compiled a collection of tools for managing Architectural Decision Records (ADRs), with a particular focus on *adr-tools*, and analyzed its strengths and weaknesses. The analysis highlighted several opportunities for improvement and potential areas where a new tool could provide enhanced functionality and usability. This section presents the concept of the Architectural Decision Guidance (ADG) tool, which was developed in response to the findings of the previous analysis. The ADG-Tool aims to address the identified weaknesses and capitalize on the opportunities to provide a more comprehensive approach to working with ADRs. The ADG-Tool is currently at the prototype stage, providing a very simple proof-of-concept that lays the foundation for more advanced functionality in the future, possibly in a follow-up thesis. The main focus of this tool lies in one of the opportunities outlined in the SWOT analysis in Section 3.2.2, namely the opportunity for a new tool to not only assist in the creation and management of ADR files, but also to act as a guidance system and assist the user in the architectural decision-making process. For this proof-of-concept, this meant adding commands similar to how *adr-tools* creates ADRs, but for creating new templates and models. This allows users to build a knowledge base of different templates used for different scenarios. This also links back to Clean Architecture, as the collection of decisions outlined in Section 4 can be used as a Clean Architecture model, and each decision in the collection is a template that can be used to create ADRs, in line with Clean Architecture principles. For starters, the prototype is just like *adr-tools*, a simple Command-line Interface (CLI) tool that can create and manipulate files. It provides a solid starting point for a tool that could significantly improve the way software architects and developers manage architectural decisions in their projects. The ADG proof-of-concept is designed to support two primary user groups; knowledge engineers and software architects/developers.

The knowledge engineer is responsible for designing and maintaining guidance models that encapsulate best practices and decision-making frameworks for specific architectural styles or project requirements. These models serve as templates that guide the decision-making process for other users. Consider the following user stories designed for the proof-of-concept outlining the usage for a knowledge engineer:

- As a knowledge engineer, I want to create a new guidance model using the command 'adg create <model>' so that it can be used to navigate the process of making architectural decisions. This allows me to encapsulate best practices and structured decision-making processes into a model that can

be reused across multiple projects. For example, I might create a guidance model that provides detailed steps for implementing decisions in alignment with Clean Architecture principles.

- As a knowledge engineer, I want to add decision points to a guidance model using the command 'adg add <model> <decision>' so that the model provides detailed guidance for specific architectural decisions. This feature enables me to ensure that the guidance model covers all necessary aspects of the decision-making process, helping users to make informed decisions. For instance, I might add decision points that address common challenges such as selecting an architectural style or deciding on a technology stack.

The software architect/developer uses the guidance models created by the knowledge engineer to make informed decisions tailored to their specific project needs. The tool helps them apply best practices and maintain consistency across their architectural decisions. Consider the following user story designed for the proof-of-concept outlining the usage for a software architect/developer:

- As a software architect/developer, I want to instantiate decisions from a guidance model using the command 'adg init <model> <path>' so that I can start making architectural decisions for my project in a structured manner. This allows me to leverage the expertise encapsulated in the guidance model, ensuring that my decisions are aligned with best practices. For example, I can instantiate a Clean Architecture model for my e-commerce platform and follow the predefined decision points to ensure a well-structured architecture.

The ADG-Tool prototype is implemented in Go [34] and provides the commands outlined in the user stories. The proof-of-concept intentionally focuses on the guidance part, and does not include basic features that are present in *adr-tools* for managing and linking individual ADRs. The primary functionalities of the tool are organized around creating and managing guidance models, which serve as structured frameworks for making architectural decisions. Go was chosen as a programming language, as it is fairly simple to create basic CLI applications, but a language like Python would have also been a good choice. During the research phase we also experimented with extending and creating new functionality for *adr-tools* using shell scripts, but in order not to exclude support for Windows we decided to build a prototype from scratch.

The first core feature of the ADG-Tool is the ability to create a new guidance model. By using the command 'adg create <model>', users can generate a new

directory within the adg-models folder. This directory becomes the root for a new guidance model, where subsequent decision points can be added. This feature is particularly useful for establishing a template that encapsulates best practices for architectural decisions, ensuring consistency and reusability across projects. For instance, when a knowledge engineer creates a model for Clean Architecture, the resulting directory structure provides a clear, organized framework for documenting and guiding architectural decisions. The following command will create a new directory called 'clean-architecture' inside the 'adg-models' directory:

```
> adg create clean-architecture
```

The tool also enables users to add decision points to an existing guidance model using the command 'adg add <model> <decision>'. This command creates a new Markdown file representing a decision point within the specified model. Each decision point can include detailed guidance on specific architectural decisions, allowing for a structured and comprehensive decision-making process. For example, a decision point might guide users through the process of deciding for a specific use case in a system using Clean Architecture, like outlined in the architectural decision in Section 4.3. The following command will create a new file called '3-select-use-cases' inside the 'adg-models/clean-architecture' folder:

```
> adg add clean-architecture 3-select-use-cases
```

Another essential feature of the ADG-Tool is the ability to initialize a model for use in a specific project. Using the command 'adg init <model> <path>', users can copy an existing model's structure into a specified path within their project directory. This feature integrates the guidance model into the development workflow, ensuring that all relevant architectural decisions are considered from the outset of the project. For example, when a software architect initializes a Clean Architecture model within their project, they can immediately begin following the predefined decision points, ensuring that their architectural decisions align with best practices. The following command will copy all decision from the 'clean-architecture' model to the specified path:

```
> adg init clean-architecture project/doc/adr
```

The Clean Decision Handbook outlined in Section 4 effectively acts as a knowledge base or template library within the ADG-Tool. When integrating the Clean Decision Handbook with the ADG-Tool, each decision outlined in the handbook corresponds to a decision point within a guidance model. This integration allows users to instantiate a project with a predefined set of architectural decision templates, ensuring that the decision-making process follows a logical, structured path. For instance, when a software architect initializes a Clean Architecture model using the ADG-Tool, the tool will generate a series of decision points based on the templates provided in the model (handbook). Each of these decision points can then be customized according to the specific needs of the project. The ADG-Tool's command 'adg init <model> <path>' facilitates this process by creating a structured environment where the architect can systematically work through each decision, filling in the details relevant to their project, while maintaining consistency with the principles outlined in the handbook. Again this is still a very rudimentary approach to a guidance model, as the prototype as of now simply copies the files from the knowledge base to the specified project path, not much different than *adr-tools* simply creates text files with a specific ADR template in your desired location. But there are several opportunities outlined in Section 3.2.2 to improve this decision guidance prototype in the future.

While the ADG-Tool prototype offers a basic yet functional approach to guide the architectural decisions, several enhancements could further elevate its utility and effectiveness. The ADG-Tool must meet several non-functional requirements to ensure its effectiveness and broad usability. It should be highly usable, providing clear and concise commands with comprehensive in-tool help documentation, and guiding users through the decision-making process with intuitive messages and prompts. Performance is another key factor, as the tool should perform actions quickly, especially when reading from or writing to disk, to ensure smooth operation in a development environment. Reliability is critical, as the tool should handle errors, providing users with meaningful error messages to help them recover from failures. In addition, the tool should be maintainable, with a well-documented and organized code base that follows coding standards, making it easy to update and integrate with future technologies. Scalability is important to accommodate different project sizes, ensuring that the tool can handle increasing numbers of ADRs as projects grow. Security measures must also be in place to protect sensitive architectural decisions and ensure data integrity. Finally, the tool must be portable, capable of running on multiple operating systems with minimal code changes, and it should minimize dependencies to simplify installation in different environments. Further opportunities mentioned in the SWOT analysis could be explored and potentially realized in a follow-up thesis, expanding the tool's capabilities beyond its

current scope. One of the most promising areas for enhancement is the development of an interactive decision-making workflow. Such a feature would guide users step-by-step through the decision-making process, providing real-time feedback and context-sensitive suggestions based on the specifics of the project. This would transform the ADG-Tool from a static file creation utility into a dynamic assistant that actively supports architects in making well-informed decisions. Another potential enhancement is the integration with popular Integrated Development Environments (IDEs). By developing plugins for IDEs like Visual Studio Code or IntelliJ IDEA, the ADG-Tool could become seamlessly integrated into the daily workflow of developers, allowing them to manage ADRs directly from their coding environment. This integration would not only streamline the decision-making process but also ensure that architectural decisions are made in close alignment with the actual codebase. Lastly, exploring the use of machine learning algorithms could open up new possibilities for the ADG-Tool. By analyzing historical data, industry best practices, and project-specific requirements, machine learning could provide decision recommendations, highlight potential risks, and even automate certain aspects of the decision-making process.

4 Clean Decision Handbook

In the previous sections, we explored the importance of managing architectural decisions and the tools available to support this aspect of software development. With a solid foundation in place, we will now focus on the application of these concepts within the Clean Architecture framework. Following the conceptual overview in Section 2, we will present a *Clean Decision Handbook* that provides insights into key architectural decisions that are critical for systems using a Clean Architecture approach. For each clean decision, an example will be provided to help architects and developers maintain adherence to Clean Architecture. But first, we will discuss the development of this handbook, considering its target audience, the scope of the decisions it covers, and the methods used to extract and organize the guidance provided. This discussion will provide insights into how the Handbook can be used in practice and its limitations. By the end of this section, readers will have a comprehensive understanding of how to apply Clean Architecture principles to their architectural decision-making processes.

The decision guidance developed in this section is primarily aimed at software architects and software developers who are responsible for making and guiding architectural decisions within their project. These individuals are often tasked with ensuring that their software systems are not only functional, but also maintainable, scalable and adaptable to future changes. The handbook is relevant to those who wish to apply the principles of Clean Architecture, which can provide significant benefits, as discussed in detail at the beginning of this thesis. The handbook is structured so that the decisions build on each other, starting with the most important decisions covering high-level architectural requirements, down to the less important decisions covering low-level implementation details. This follows the common theme in Clean Architecture of high-level policies and low-level details. The idea is that a team can follow these decisions more or less step by step to design a system, whether it is a completely new project from scratch, or an existing project that is being rebuilt in a Clean Architecture style. Therefore, the decisions covered in this handbook represent a selected subset of the overall architectural scope. While they address many of the most critical aspects of getting started with Clean Architecture, they do not cover every possible decision an architect might face. Instead, it focuses on decisions that have the greatest impact on maintaining the integrity of Clean Architecture principles, such as defining the initial architecture layout with its entities and use cases, choosing the right patterns, and managing dependencies to frameworks and external systems. The guide is intended to be a starting point rather than a comprehensive guide. Architects are encouraged to adapt and extend the guidance provided, taking into account the specific needs

and constraints of their projects. The templates used throughout the handbook are illustrative and, while they cover common challenges, they may not address all the unique situations that may arise when faced with a particular decision. They are intended to help architects and developers understand the implications of each decision and how it fits with the overall architectural objectives.

We derived the architectural choices presented in this handbook from a detailed study of Robert C. Martin's book "Clean Architecture" [1], supplemented by practical insights from the area of architectural decision-making covered in the previous sections, as well as our own professional experience of working with software using a Clean Architecture style. The process involved analyzing the key principles underlying Clean Architecture, as discussed in Section 2.1, or the layered structure with its emphasis on separation of concerns and the dependency rule, as discussed in Section 2.2. An overview of the result can be seen in Figure 10, which illustrates a clear flow of decisions to be made. It is important to note that the decisions enclosed in the larger box cover decisions relating to the architectural layers and should therefore be made in the order shown. However, the bottom three decisions can be made after any of the decisions in the big box if necessary, indicated by an arrow going from the big box, and there is an additional fourth placeholder box representing other decisions that may need to be made.

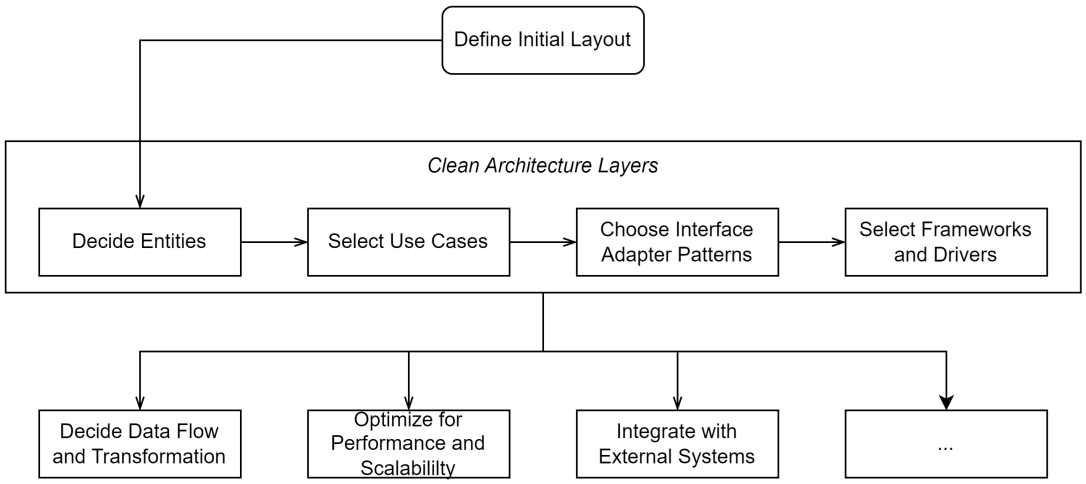

Figure 10: Clean Decisions

4.1 Defining the Architecture's Initial Layout

One of the earliest and most critical decisions in applying Clean Architecture is to define the initial layout of the architecture. This decision is about committing to the overarching structure and principles of Clean Architecture, which fundamentally dictate how the entire system will be organized and maintained over time. By choosing Clean Architecture, you are choosing to enforce a layered approach where the core business logic, encapsulated in entities, is isolated from external concerns by layers such as use cases, interface adapters, frameworks and drivers. [1, Chapter 22]

There are several key aspects to this decision. First, it requires a commitment to the Clean Architecture model itself. This choice is important because it defines the basic structure of the system and ensures that responsibilities are clearly separated across the different layers. Secondly, defining the initial layout also involves determining how to implement the prescribed layered structure within the specific context of your project. While Clean Architecture provides a general framework — entities, use cases, interface adapters, frameworks and drivers — there is flexibility in how these layers are applied. Depending on the complexity and needs of the project, the layers can be expanded or adapted [1, p. 205], but the core principle remains the same; to keep the business logic isolated from external systems, thus maintaining the flexibility of the architecture.

This fundamental choice influences how all subsequent decisions are made, ensuring that the architecture remains robust and adaptable over time. It is about laying the groundwork for how the components of the system will interact, ensuring that the core business logic remains unaffected by changes in external systems.

Example: In our e-commerce platform scenario, defining the initial layout of the architecture would involve committing to Clean Architecture and deciding that the system will be structured into the four primary layers: Entities, Use Cases, Interface Adapters, and Frameworks and Drivers. This structure will guide how the codebase is organized, ensuring that the core business logic, such as how orders are processed, is insulated from changes in external systems such as database technology or the UI framework. For this first decision, we will provide a completed ADR of this example (see Figure 12) based on the template (see Figure 11) to get a first idea. For the following decisions, only the template will be provided.

Title

```
Defining the Architecture's Initial Layout for <project>
```

Context

```
We are developing a project that <context>.  It needs to
be maintainable, adaptable to future changes, <additional
requirements>.
```

Decision

```
We have decided to adopt the Clean Architecture model as the
foundational structure and the system will be organized into <n>
distinct layers:
- Entities:  Representing the core business objects and rules, such
as <some examples>.
- Use Cases:  Managing the application-specific business logic and
workflows, such as <some examples>.
- Interface Adapters:  Handling the interactions between the
internal layers and external systems, such as <some examples>.
- Frameworks and Drivers:  The outermost layer includes frameworks,
such as <some examples>.
- <additional layers>
```

Status

```
Proposed
```

Consequences

```
- Positive:  This layered approach will ensure that the core
business logic remains insulated from changes in external systems,
maintaining the system's flexibility and adaptability over time.
- Negative:  Initial setup and adherence to the Clean Architecture
model may require more upfront effort and learning for the
development team.  The strict separation of layers may introduce
additional complexity in managing dependencies and ensuring that
communication between layers is efficient and well-structured.
- <additional consequences>
```

Figure 11: Template Architectural Decision Record (Nygard) for defining the initial Clean Architecture layout, with placeholders denoted by angle brackets (<>).

Title

Defining the Architecture's Initial Layout for the E-Commerce Platform

Context

We are developing a e-commerce platform that provides users a way to order and buy our products online. It needs to be maintainable, adaptable to future changes, be able to integrate with various payment gateways, and supporting both web and mobile interfaces.

Decision

We have decided to adopt the Clean Architecture model as the foundational structure and the system will be organized into 4 distinct layers:
- Entities: Representing the core business objects and rules, such as ‘Payment‘, ‘Customer‘, and ‘Product‘.
- Use Cases: Managing the application-specific business logic and workflows, such as ‘RegisterCustomer‘ and ‘ProcessPayment‘.
- Interface Adapters: Handling the interactions between the internal layers and external systems, such as data translation and communication with databases, APIs, and user interfaces.
- Frameworks and Drivers: The outermost layer includes frameworks, such as the web server, database technology, and UI frameworks.

Status

Accepted

Consequences

- Positive: This layered approach will ensure that the core business logic remains insulated from changes in external systems, maintaining the system's flexibility and adaptability over time and allows to implement both web and mobile interfaces without affecting the core business logic.
- Negative: Initial setup and adherence to the Clean Architecture model may require more upfront effort and learning for the development team. The strict separation of layers may introduce additional complexity in managing dependencies and ensuring that communication between layers is efficient and well-structured.

Figure 12: Example Architectural Decision Record (Nygard) for defining the initial layout of an E-commerce shop, using the template from Figure 11, filling in the placeholders.

4.2 Deciding the Boundaries for Entities

Entities represent the core business rules of the system and are the most stable and reusable elements in Clean Architecture. Deciding how to define the boundaries of these entities is critical, as it determines how well they can encapsulate critical business logic while remaining independent of external factors such as databases or user interfaces. This decision involves identifying the key business concepts that should be modelled as entities, as opposed to use cases, and ensuring that these entities are designed to work independently of how data is stored, retrieved or presented to users. Entities should not contain complex application-specific functionality. [1, p. 190, 204]. This decision involves defining the boundaries of the entity layer by analyzing the core business model and defining which entities the application needs to fulfil the business rules. [1, Chapter 17, 18] Deciding these boundaries closely aligns with the principles of Domain-Driven Design (DDD), which emphasizes that entities should represent key business concepts within a bounded context, ensuring that they are cohesive and encapsulate only the necessary business logic. Vaughn Vernon's "Implementing Domain-Driven Design" provides further insights into how to define these boundaries effectively. [35] A decision here could be as simple as defining and clarifying one or more entities that encompass a particular business rule required by the system. A template for making this type of decision can be seen in Figure 13.

Example: In our e-commerce platform, the 'Payment' entity would encapsulate information about a specific payment, such as the amount, currency, and status that a customer's payment has been successfully processed. However, the Payment entity should not process a transaction using implementations of payment processing systems (e.g. PayPal). Instead, these functionalities and dependencies should be handled by other layers (e.g. use cases, interface adapters) to ensure that the 'Payment' entity remains consistent and reusable across different applications or contexts, since the information about a payment is generally the same and independent of how it has been processed by different applications.

Title

Defining the Boundaries for Entities in <project> encapsulating <business rule>

Context

We are designing the core business rules for <project>, which need to be stable and reusable across different applications and independent of external frameworks or systems. The entities must encapsulate <business rule> while remaining isolated from external systems.

Decision

We have decided to define the following entity/entities to represent the core business rules:
- <entity 1>: Represents <key business concept>, responsible for <specific business logic>.
- <more entities>

Status

Proposed

Consequences

- Positive: The entities will be stable and reusable, allowing for consistency across different applications and contexts.
- Negative: The strict independence of entities from external systems may require additional effort to ensure proper encapsulation and separation of concerns.
- <additional consequences>

Figure 13: Template Architectural Decision Record (Nygard) for deciding the boundaries of one (or multiple) entities, with placeholders denoted by angle brackets (<>).

4.3 Selecting Use Cases

Use cases in Clean Architecture represent the application-specific business rules that dictate how the system behaves in response to certain actions or events. Use cases should be designed to manage specific workflows or processes, ensuring that they are focused and free from unnecessary dependencies on the outside world. [1, p. 148, 191-193, 204] A decision involves deciding on a specific use case, outlining the workflow required by the application and the actions that will be performed by the use case. In practice, many use cases are required for an application, and therefore several decisions are recorded for selecting the many different use cases. A template for making this type of decision can be seen in Figure 14.

Example: In our e-commerce platform, the 'RegisterCustomer' use case would be responsible for managing the entire workflow of creating an account for a new customer. This includes validating the details provided by the user upon registration, creating a new 'Customer' entity and updating our customer database and notifying the UI that the registration was successful (using abstractions and dependency injections to comply with the dependency rule). By isolating this logic within a dedicated use case, the system can adapt to changes, such as adding additional validation, without changing the core 'customer' entity. This separation ensures that each aspect of the process is handled consistently and that the use case can be reused or extended as required.

Title

Selecting the Use Case for <specific workflow> in <project>

Context

We need to implement a use case that handles the <specific workflow>. This use case will manage the application-specific business logic associated with <specific action or event> and must remain independent of external systems.

Decision

We have decided to structure the <use case> to handle the <specific workflow> by performing the following actions:
- <action 1>: <describe specific action the use case will perform>.
- <more actions>

Status

Proposed

Consequences

- Positive: The use case will provide a clear and isolated management of the workflow, making it adaptable to future changes.
- Negative: Additional interfaces may need to be developed to ensure that the use case remains decoupled from external systems.
- <additional consequences>

Figure 14: Template Architectural Decision Record (Nygard) to select and implement a specific use case, with placeholders denoted by angle brackets (<>).

4.4 Choosing Patterns for Interface Adapters

Interface adapters are responsible for translating data between the internal layers of the system (entities and use cases) and external systems (such as the user interface or databases) [1, p. 205]. Deciding which patterns to use for these adapters is critical, as it directly affects the system's ability to maintain a clean separation between its core logic and the external environment. Common patterns include the Repository pattern for data access and the Presenter pattern for handling UI interactions. The goal is to decouple the business logic from volatile external systems, while maintaining the flexibility and maintainability of the architecture. [1, Chapter 23] A template for making this type of decision can be seen in Figure 15.

Example: In our e-commerce platform, since we need to interact with a database, the repository pattern might be best suited to abstract the interaction between the 'UpdateInventory' use case and a MongoDB database. Instead of the use case interacting directly with the database, it interacts with a ProductRepository interface, which then manages the database operations. Similarly, a controller pattern could be used to handle HTTP requests, such as a request to register a new account, from the web application, converting these requests into actions that the use cases can process. This approach ensures that changes to the database or user interface do not require changes to the core business logic.

Title

```
Choosing the <pattern> in <project>
```

Context

```
We need to choose a pattern for <detail> to ensure effective
translation of data between <specific use case> and <external
system> to prevent direct dependencies to external systems.
```

Decision

```
We have decided to implement the <pattern> for <detail>.
This pattern will:
- <responsibility 1>:  <describe how the pattern will handle
specific interaction or data translation>.
- <more responsibilities>
```

Status

```
Proposed
```

Consequences

```
- Positive:  The use of <pattern> will ensure that changes to
<external system> do not impact <core business logic>.
- Negative:  Implementing <pattern> may introduce additional
complexity in managing the interactions between layers.
- <additional consequences>
```

Figure 15: Template Architectural Decision Record (Nygard) to choose a pattern for the interface adapters, with placeholders denoted by angle brackets (<>).

4.5 Selecting Frameworks and Drivers

While Clean Architecture advises against letting specific technologies dictate the overall system architecture, the choice of frameworks and drivers for the outermost layer is still a critical decision. This decision involves selecting technologies that are best suited to the current needs of the project, while ensuring that these technologies can be replaced or upgraded without affecting the core architecture. Technologies chosen for this layer should be focused on external concerns like databases, user interfaces, and third-party services. The core business logic should never directly depend on these technologies, preventing tight coupling. The decision should be made with replaceability in mind, allowing the system to evolve as technologies change without disrupting the core business logic. [1, Chapter 30-32] A template for making this type of decision can be seen in Figure 16.

Example: For our e-commerce platform, we might initially choose PostgreSQL as our database technology and React as our front-end framework. These technologies are chosen based on their current fit with the needs of the project. However, by restricting them to the framework and driver layer, we ensure that the system remains flexible. In the future, if we decide to switch to a NoSQL database or adopt a different front-end framework, we can do so without having to change the core business logic or use cases, thus maintaining the adaptability of the system.

Title

```
Selecting the <technology> for <project>
```

Context

```
For <specific functionality>, we need to select a technology
that meets current project requirements in terms of
<performance/scalability/availability/etc.>, and can be easily
replaced or upgraded in the future without affecting the core
business logic by isolating it within the Frameworks and Drivers
layer.
```

Decision

```
We have chosen <technology> for <functionality> within the
Frameworks and Drivers layer.  The decision was based on the
following criteria:
- <reason 1>:  <describe why this technology is suitable>.
- <more reasons>
```

Status

```
Proposed
```

Consequences

```
- Positive:  The system will remain adaptable to future technology
changes, maintaining the core architecture's stability.
- Negative:  The initial integration may require additional effort
to ensure the technology remains confined to the Frameworks and
Drivers layer.
- <additional consequences>
```

Figure 16: Template Architectural Decision Record (Nygard) to select a framework or driver, with placeholders denoted by angle brackets (<>).

4.6 Deciding on Data Flow and Transformations

The flow and transformation of data between layers is another critical decision in Clean Architecture. This involves choosing the appropriate data formats at each layer boundary and deciding how data should be transformed as it moves between layers. The objective is to maintain clarity and consistency in how data is handled, ensuring that transformations do not introduce unnecessary complexity or tightly couple different parts of the system. The data flow should be designed to support the separation of concerns, with each layer only dealing with data in its relevant context. Typically this data consist of simple data structures, like simple data transfer objects, a hashmap containing the important information, or even just arguments in function calls. The idea is to not pass entire entity objects or database rows and never violate the dependency rule. [1, p. 207] A template for making this type of decision can be seen in Figure 17.

Example: When a customer places an order on our e-commerce platform, the HTTP request data received by the web server would first be transformed into a domain-specific format before being passed to the PlaceOrder use case. This ensures that the use case operates with data in a format that aligns with the business logic. Once the order is processed, the resulting data would be transformed back into a format suitable for the user interface, allowing the response to be presented to the customer. By handling data transformations at the boundaries between layers, we maintain the integrity and independence of each layer.

Title

```
Deciding on Data Flow and Transformation for <interaction>
```

Context

```
We need to define how data will flow and be transformed during
<interaction>.  The data must be appropriately formatted at each
layer boundary, maintaining clarity and consistency in the system.
```

Decision

```
We have decided that during <interaction>, data will flow from
<source layer> to <target Layer> and will be transformed as follows:
- <transformation 1>:  Data will be transformed from <source format>
to <target format> when moving from <source layer> to <target
layer>.
- <more transformations>
```

Status

```
Proposed
```

Consequences

```
- Positive:  This clear data flow reduces the risk of errors and
keeps layers decoupled and independent.
- Negative:  Implementing these transformations may introduce
complexity in ensuring consistency across layers.
- <additional consequences>
```

Figure 17: Template Architectural Decision Record (Nygard) for deciding data flow and transformations, with placeholders denoted by angle brackets (<>) Template ADR (Nygard).

4.7 Optimizing for Performance and Scalability

Optimization is not necessarily one of the first things to consider when designing the architecture of a system, and Martin even refers to it as a low-level architectural concern, since with a clean architecture it can be completely encapsulated and separated from the business rules [1, p. 281], which is why it is also one of the later decisions in this collection. As Kent Beck put it (quoted by Martin in the book) "First make it work. Then make it right. Then make it fast". [1, p. 258] Nevertheless, decisions about optimizing for performance and scalability are important, especially in systems that are expected to handle large volumes of data or high transaction rates. The challenge is to achieve optimization without compromising the flexibility of the architecture. This involves choosing strategies that allow the system to scale horizontally or vertically, or perform faster, while maintaining a clean separation between tiers. The idea is that the core business and application logic does not need to worry about performance or scalability. For example, if a use case needs to call a computationally expensive algorithm, the implementation should be extracted as a service and placed in the framework layer. This allows the use case to call the service interface, while we can adapt and change or even replace the algorithm to optimize it, without affecting the use case itself. Another example of a common optimization technique is database partitioning, and since the database is a detail belonging to the outermost layer [1, p. 277], optimizing it will not affect the core business logic. A template for making this type of decision can be seen in Figure 18.

Example: For scalability in our e-commerce platform, we might decide to implement a queue-based order processing system to ensure greater reliability and fault tolerance during high traffic periods. This will ensure that the system can handle high volumes of traffic without overwhelming the server, although it may result in slight delays in order confirmation on the user side. In addition, caching could be implemented to reduce the load on the database during peak traffic periods. Both of these optimization strategies are isolated in the outer layer so that the system can scale effectively while maintaining a clean separation between the core business logic and these optimization implementations.

Title

```
Implementing <optimization strategy> for <workflow>
```

Context

```
We need to address optimize <workflow> to ensure the system can
<requirements> while maintaining Clean Architecture principles.
This optimization must be confined to the outer layers to avoid
compromising the flexibility and independence of the core business
logic.
```

Decision

```
We have decided to implement <optimization strategy> to optimize
<workflow>.  This strategy will involve:
- <action 1>:  Implementing <specific optimization strategy> by
<technique to isolate from business logic>.
- <more actions>.
```

Status

```
Proposed
```

Consequences

```
- Positive:  The system will be optmized based on <benefits> while
maintaining its clean architectural structure.
- Negative:  The chosen strategy may introduce complexity in
ensuring the system remains decoupled and maintainable.
- <additional consequences>
```

Figure 18: Template Architectural Decision Record (Nygard) for optimizing performance and scalability, with placeholders denoted by angle brackets (<>).

4.8 Integrating with External Systems

Integration with external systems, such as third-party APIs or systems from external companies, is a common requirement in modern software systems. However, these integrations must be handled carefully to avoid introducing tight coupling or compromising the flexibility of the architecture. A change by an external system should not cause our system to break, which means that we want to have an asymmetric relationship with any external system. [1, p. 172] Asymmetric means that the external system (or rather the implementation of it in our framework layer) depends on the use cases of our application and not vice versa. If there is a change in a third-party API, the inner layers are unaffected. This decision is similar to the technology selection decision, and involves selecting an external system that can be replaced or upgraded without affecting the core architecture. Without proper setup in the interface adapter layer, we would not be able to integrate such external systems without creating dependencies, so this decision is one of the last to be made. A template for making this type of decision can be seen in Figure 19.

Example: For payment processing in our e-commerce platform, we may want to use third-party payment services such as PayPal or Stripe. By setting up a gateway interface properly, we are able to isolate the integration details and the core logic for processing orders remains the same regardless of which payment service is used. This approach ensures that the system can easily switch to a different payment provider in the future without requiring changes to the core business logic or use cases.

Title

```
Integrating <external system> with using <integration approach>
```

Context

```
We need to integrate <external System> with while ensuring that the
core business logic remains independent of it.  The integration must
be managed to avoid tight coupling and maintain the architecture's
flexibility.
```

Decision

```
We have decided to integrate <external system> using <integration
approach> to manage interactions.  This approach includes:
- <action 1>:  <describe how the external system will be integrated
without affecting core logic>.
- <more actions>
```

Status

```
Proposed
```

Consequences

```
- Positive:  The core business logic will remain decoupled from
external systems, ensuring long-term adaptability.
- Negative:  Implementing this integration approach may require
additional effort to maintain consistent communication and
interaction.
- <additional consequences>
```

Figure 19: Template Architectural Decision Record (Nygard) for for integrating external systems, with placeholders denoted by angle brackets (<>).

5 Conclusion

This thesis has explored the management of architectural decisions in Clean Architecture, presenting both theoretical insights and practical solutions for handling architectural decisions in modern software systems. Clean Architecture's emphasis on maintaining flexibility and separation of concerns makes it an excellent candidate for systems that require long-term adaptability and maintainability. The core of this work has revolved around understanding the crucial role of architectural decisions and the tools and processes required to manage them effectively.

Section 2 of the thesis began by introducing the basic software design principles and the four layers (Entities, Use Cases, Interface Adapters, and Frameworks and Drivers) of Clean Architecture. A case study of an e-commerce platform was used to demonstrate the practical application of these layers and to help the reader understand how Clean Architecture principles are applied in real-world scenarios.

In Section 3, the thesis explored the process of documenting architectural decisions through the use of Architectural Decision Records (ADRs). This section covered the importance of documenting decisions for long-term maintenance, transparency and decision traceability. The introduction of decision documentation formats such as the Nygard format, Y-statement and MADR provided a structured way of capturing the rationale and consequences of architectural decisions. The discussion highlighted the strengths and weaknesses of each format, supported by specific examples tailored to the e-commerce platform case study, such as the choice between microservices and monolithic architecture, or the selection of technology frameworks.

Building on this foundation, Section 4 presented the Clean Decision Handbook, a key contribution of this thesis. This handbook provides a structured framework for managing Clean Architecture decisions, with templates tailored for different types of decisions. Each decision was carefully categorized, including decisions related to defining the architecture layout, selecting use cases, choosing patterns for interface adapters, and addressing performance and scalability concerns. The templates provided actionable guidance for architects and developers to make informed, consistent decisions within the context of Clean Architecture. The handbook serves as a practical tool for applying the theoretical concepts discussed in previous sections to ensure that architectural decisions are made in accordance with Clean Architecture principles. Looking forward, extending the Clean Decision Handbook to include a more comprehensive set of architectural decisions for Clean Architecture could increase its utility for larger, more complex systems. In addition, explor-

ing the application of these principles and developing a more general handbook for different architectural styles beyond Clean Architecture could provide valuable insights for broader use cases.

Section 3.2.3 discussed the ADG-Tool prototype, which was developed to address some of the gaps identified in existing ADR tools. Although the prototype is still in its early stages, it provides a starting point for more advanced tools in the future. The ADG-Tool is designed to help architects and developers create, manage and track architectural decisions using predefined templates and structured processes. While the prototype provides basic functionality, such as creating and initializing decision models, the potential for future enhancements is significant. Future iterations of the tool could include interactive decision workflows, real-time collaboration and integration with development environments, as described in Section 3.2.3. This prototype lays the groundwork for future research and development in this domain. In a follow-up thesis, the tool could be further developed to include advanced decision analysis, more robust collaboration features, and real-time impact analysis of architectural decisions. Integration with machine learning could provide decision recommendations based on historical data, allowing the tool to become a dynamic assistant in the decision-making process.

In conclusion, this thesis has contributed both theoretical knowledge and practical tools for managing architectural decisions within Clean Architecture. By effectively documenting these decisions and providing tools to guide the decision-making process, architects and developers can ensure that their systems remain flexible, maintainable and adaptable to future challenges. The work done here is a step towards a more structured and traceable approach to software architecture, with the potential for significant progress in future research and tool development.

References

[1] R. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2018.

[2] A. Cockburn, "Hexagonal Architecture," 2005, accessed: September 6, 2024. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/

[3] J. Palermo, "The Onion Architecture : part 1," 2008, accessed: September 6, 2024. [Online]. Available: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[4] R. Martin, "The Clean Architecture," 2012, accessed: September 6, 2024. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[5] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," 2004, accessed: September 6, 2024. [Online]. Available: https://www.martinfowler.com/articles/injection.html?ref=danyow.net

[6] ——, "Gateway," 2021, accessed: September 6, 2024. [Online]. Available: https://martinfowler.com/articles/gateway-pattern.html

[7] ——, "Repository," 2003, accessed: September 6, 2024. [Online]. Available: https://martinfowler.com/eaaCatalog/repository.html

[8] ——, "Front Controller," 2003, accessed: September 6, 2024. [Online]. Available: https://martinfowler.com/eaaCatalog/frontController.html

[9] ——, "Supervising Controller," 2006, accessed: September 6, 2024. [Online]. Available: https://martinfowler.com/eaaDev/SupervisingPresenter.html

[10] O. Zimmermann, "From Architectural Decisions to Design Decisions," 2021, accessed: September 6, 2024. [Online]. Available: https://medium.com/olzzio/from-architectural-decisions-to-design-decisions-f05f6d57032b

[11] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, 2005.

[12] M. Nygard, "Documenting Architecture Decisions," 2011, accessed: September 6, 2024. [Online]. Available: https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions

[13] ADR GitHub Organization, "Architectural Decision Records," accessed: September 6, 2024. [Online]. Available: https://adr.github.io

[14] O. Zimmermann, "Making Architectural Knowledge Sustainable – The Y-Approach, Industrial Practice Report and Outlook," in *Proceedings of the 9th Annual SATURN Conference*, 2012.

[15] ——, "Y-Statements," 2021, accessed: September 6, 2024. [Online]. Available: https://medium.com/olzzio/y-statements-10eb07b5a177

[16] ——, "Architectural Decisions — The Making Of," 2021, accessed: September 6, 2024. [Online]. Available: https://www.ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html

[17] ADR GitHub Organization, "Markdown Architectural Decision Records," accessed: September 6, 2024. [Online]. Available: https://adr.github.io/madr/

[18] O. Kopp, A. Armbruster, and O. Zimmermann, "Markdown Architectural Decision Records: Format and Tool Support," in *Proceedings of the 10th Central European Workshop on Services and their Composition*, 2018.

[19] ADR GitHub Organization, "Markdown Architectural Decision Records Examples," accessed: September 6, 2024. [Online]. Available: https://adr.github.io/madr/examples.html

[20] ——, "GitHub repository: adr/madr," accessed: September 6, 2024. [Online]. Available: https://github.com/adr/madr/

[21] O. Zimmermann, "The Markdown ADR (MADR) Template Explained and Distilled," 2022, accessed: September 6, 2024. [Online]. Available: https://ozimmer.ch/practices/2022/11/22/MADRTemplatePrimer.html

[22] ADR GitHub Organization, "GitHub repository: adr/adr-log," accessed: September 6, 2024. [Online]. Available: https://github.com/adr/adr-log

[23] A. Wilson, "GitHub repository: mrwilson/adr-viewer," accessed: September 6, 2024. [Online]. Available: https://github.com/mrwilson/adr-viewer

[24] ADR GitHub Organization, "GitHub repository: adr/e-adr," accessed: September 6, 2024. [Online]. Available: https://github.com/adr/e-adr

[25] C. Sprayberry, "GitHub repository: cspray/architectural-decision," accessed: September 6, 2024. [Online]. Available: https://github.com/cspray/architectural-decision

[26] N. Pryce, "GitHub repository: nypryce/adr-tools," accessed: September 6, 2024. [Online]. Available: https://github.com/npryce/adr-tools

[27] Canpolat, "GitHub repository: canpolat/talo," accessed: September 6, 2024. [Online]. Available: https://github.com/canpolat/talo

[28] T. Vaillant, "GitHub repository: thomvaill/log4brains," accessed: September 6, 2024. [Online]. Available: https://github.com/thomvaill/log4brains

[29] E. Sciara, "GitHub repository: opinionated-digital-center/pyadr," accessed: September 6, 2024. [Online]. Available: https://github.com/opinionated-digital-center/pyadr

[30] ADR GitHub Organization, "GitHub repository: adr/adr-manager," accessed: September 6, 2024. [Online]. Available: https://github.com/adr/adr-manager

[31] Coding Friends GmbH, "Loqbooq," accessed: September 6, 2024. [Online]. Available: https://loqbooq.app/how-it-works

[32] Loomio Cooperative Limited, "Loomio," accessed: September 6, 2024. [Online]. Available: https://www.loomio.com

[33] E. Choice, "The Analytic Hierarchy Process: Structured Decisions," accessed: September 6, 2024. [Online]. Available: https://www.expertchoice.com/ahp-software

[34] R. Griesemer, R. Pike, and K. Thompson, "Go programming language," accessed: September 6, 2024. [Online]. Available: https://go.dev

[35] V. Vernon, *Implementing Domain-driven Design*. Addison-Wesley Professional, 2013.