RCMOS – Revista Científica Multidisciplinar O
Saber. ISSN: 2675-9128. São Paulo - SP.
Ano VII, v.1 2026 | submissão: 11/07/2026 | aceito: 08/07/2026 | publicação: 14/07/2026

Clean Architecture en el Desarrollo de Software Moderno: ¿Aliada o Villana?

Clean Architecture no Desenvolvimento de Software Moderno: Aliada ou Vilã?
Clean Architecture in Modern Software Development: Ally or Villain?

Wanderley Junior Alves Trindade
ORCID: 0009-0005-7962-272X
Lattes: http://lattes.cnpq.br/1193694031486153

Resumen: La creciente complejidad de los sistemas modernos ha hecho que las decisiones arquitectónicas sean fundamentales para garantizar la organización, la calidad y la evolución del software. En este contexto, la Clean Architecture propone separar las reglas de negocio de los detalles tecnológicos, como frameworks, bases de datos, interfaces y servicios externos. El objetivo general de este trabajo fue analizar críticamente la aplicación de la Clean Architecture en el desarrollo de software moderno, considerando sus principios, beneficios, limitaciones e impactos en la calidad, el mantenimiento y la evolución de los sistemas. Metodológicamente, se realizó una investigación bibliográfica de enfoque cualitativo y carácter descriptivo y exploratorio, basada en artículos científicos en lengua portuguesa publicados en los últimos diez años y disponibles gratuitamente en revistas y bases académicas. Los resultados demostraron que la Clean Architecture puede favorecer la mantenibilidad, la capacidad de prueba, la flexibilidad y la longevidad de los sistemas, principalmente en proyectos complejos, duraderos y sujetos a cambios frecuentes. Sin embargo, su implementación también puede aumentar el número de capas, clases, interfaces y abstracciones, elevando los costes y la complejidad inicial, especialmente en aplicaciones pequeñas o con reglas simples. Se concluyó que la Clean Architecture no puede considerarse exclusivamente una aliada o una villana, pues su efectividad depende de las características del proyecto, de la experiencia del equipo y de la aplicación proporcional de sus principios.
Palabras clave: Arquitectura de software. Clean Architecture. Desarrollo de software.

Resumo: A crescente complexidade dos sistemas modernos tornou as decisões arquiteturais fundamentais para garantir a organização, a qualidade e a evolução do software. Nesse contexto, a Clean Architecture propõe a separação entre as regras de negócio e os detalhes tecnológicos, como frameworks, bancos de dados, interfaces e serviços externos. O objetivo geral deste trabalho foi analisar criticamente a aplicação da Clean Architecture no desenvolvimento de software moderno, considerando seus princípios, benefícios, limitações e impactos na qualidade, na manutenção e na evolução dos sistemas. Metodologicamente, realizou-se uma pesquisa bibliográfica, de abordagem qualitativa e caráter descritivo e exploratório, baseada em artigos científicos em língua portuguesa, publicados nos últimos dez anos e disponíveis gratuitamente em periódicos e bases acadêmicas. Os resultados demonstraram que a Clean Architecture pode favorecer a manutenibilidade, a testabilidade, a flexibilidade e a longevidade dos sistemas, principalmente em projetos complexos, duradouros e sujeitos a mudanças frequentes. Entretanto, sua implementação também pode aumentar o número de camadas, classes, interfaces e abstrações, elevando os custos e a complexidade inicial, especialmente em aplicações pequenas ou com regras simples. Concluiu-se que a Clean Architecture não pode ser considerada exclusivamente uma aliada ou uma vilã, pois sua efetividade depende das características do projeto, da experiência da equipe e da aplicação proporcional de seus princípios.
Palavras-chave: Arquitetura de software. Clean Architecture. Desenvolvimento de software.

Abstract: The increasing complexity of modern systems has made architectural decisions fundamental to ensuring software organization, quality, and evolution. In this context, Clean

Architecture proposes a separation between business rules and technological details, such as frameworks, databases, interfaces, and external services. The overall objective of this study was to critically analyze the application of Clean Architecture in modern software development, considering its principles, benefits, limitations, and impacts on system quality, maintenance, and evolution. Methodologically, a literature review was conducted using a qualitative, descriptive, and exploratory approach, based on Portuguese-language scientific articles published within the last ten years and freely available in journals and academic databases. The results demonstrated that Clean Architecture can enhance system maintainability, testability, flexibility, and longevity, particularly in complex, long-lasting projects subject to frequent changes. However, its implementation can also increase the number of layers, classes, interfaces, and abstractions, thereby raising initial costs and complexity, especially in small applications or those with simple rules. It was concluded that Clean Architecture cannot be viewed solely as an ally or a villain, as its effectiveness depends on project characteristics, team experience, and the proportionate application of its principles.
Keywords: Software architecture. Clean Architecture. Software development.

1 Introducción

El desarrollo de software moderno se ha caracterizado por el aumento de la complejidad de los sistemas, la necesidad de actualizaciones frecuentes y la búsqueda de aplicaciones capaces de acompañar los cambios tecnológicos y organizacionales. En este contexto, la arquitectura de software desempeña un papel fundamental, pues define la forma en que los componentes de un sistema se estructuran, organizan y relacionan. Una arquitectura adecuada puede facilitar el mantenimiento, la realización de pruebas, la sustitución de tecnologías y la evolución del software, mientras que las decisiones arquitectónicas inadecuadas pueden generar dependencias excesivas, dificultades de comprensión y un aumento de los costes de desarrollo.

Entre los enfoques utilizados para organizar sistemas de software destaca la Clean Architecture, o Arquitectura Limpia, propuesta con la finalidad de separar las reglas de negocio de los elementos externos de la aplicación, como bases de datos, interfaces gráficas, frameworks y servicios de terceros. Su estructura busca garantizar que las partes más importantes del sistema no dependan directamente de tecnologías específicas, permitiendo que los cambios externos se realicen con menor impacto sobre la lógica central del software. Esta propuesta se relaciona con principios como la separación de responsabilidades, la inversión de dependencias, el bajo acoplamiento y la alta cohesión.

La aplicación de la Clean Architecture se ha asociado con beneficios como una mayor mantenibilidad, capacidad de prueba, flexibilidad, escalabilidad y facilidad de evolución de los sistemas. Al organizar el código en capas independientes, esta arquitectura puede contribuir a que los equipos comprendan mejor las responsabilidades de cada componente y realicen cambios sin comprometer otras partes de la aplicación. Además, la independencia respecto de

frameworks y bases de datos puede favorecer la sustitución de tecnologías y la reutilización de las reglas de negocio en diferentes contextos.

A pesar de estos beneficios, la adopción de la Clean Architecture también puede presentar limitaciones. En sistemas pequeños, proyectos de corta duración o aplicaciones con reglas de negocio poco complejas, la creación de diversas capas, interfaces, abstracciones y estructuras adicionales puede aumentar el volumen de código y dificultar el desarrollo. La aplicación rígida de sus principios también puede provocar complejidad innecesaria, elevar el tiempo de implementación y exigir mayores conocimientos técnicos del equipo. Por lo tanto, su utilización no debe considerarse automáticamente adecuada para todos los proyectos; es necesario analizar el contexto, los requisitos y las características del sistema.

Ante esta dualidad, se establece la siguiente pregunta de investigación: ¿en qué medida la aplicación de la Clean Architecture contribuye a la calidad y la evolución del software moderno y en qué situaciones su adopción puede generar complejidad, costes y dificultades innecesarias en el proceso de desarrollo? A partir de esta pregunta, el estudio busca comprender si la Clean Architecture debe considerarse una aliada del desarrollo moderno o si, cuando se emplea de manera inadecuada, puede convertirse en un factor perjudicial para los proyectos.

La realización de este trabajo se justifica por la necesidad de comprender críticamente las consecuencias de adoptar modelos arquitectónicos en el desarrollo de software. Aunque la Clean Architecture se presenta con frecuencia como una solución para problemas relacionados con el mantenimiento y la organización de los sistemas, su implementación exige planificación, experiencia y análisis de las necesidades reales del proyecto. Así, discutir sus ventajas y limitaciones puede ayudar a estudiantes, desarrolladores, arquitectos de software y organizaciones a tomar decisiones más adecuadas, evitando tanto la construcción de sistemas excesivamente acoplados como el uso de estructuras complejas sin una necesidad concreta.

El objetivo general de este trabajo consiste en analizar críticamente la aplicación de la Clean Architecture en el desarrollo de software moderno, considerando sus principios, beneficios, limitaciones e impactos en la calidad, el mantenimiento y la evolución de los sistemas. Para alcanzar este objetivo, se definieron los siguientes objetivos específicos: presentar los principales conceptos, principios y capas que componen la Clean Architecture, destacando su relación con la separación de responsabilidades y la independencia entre los componentes del software; identificar los beneficios proporcionados por el uso de la Clean Architecture, especialmente en relación con la mantenibilidad, la capacidad de prueba, la escalabilidad, la flexibilidad y la longevidad de los sistemas; e investigar las limitaciones y los desafíos asociados con su implementación, evaluando en qué contextos su adopción puede

generar complejidad excesiva, aumento de costes y dificultades durante el proceso de desarrollo.

En cuanto a los procedimientos metodológicos, el estudio se desarrolló mediante una investigación bibliográfica de enfoque cualitativo y carácter descriptivo y exploratorio. El estudio teórico consideró artículos científicos publicados en lengua portuguesa durante los últimos diez años, disponibles gratuitamente en bases académicas y revistas de acceso abierto, incluidos materiales localizados en Google Académico. Se seleccionaron trabajos relacionados con la Clean Architecture, los principios SOLID, la inversión de dependencias, la modularidad, la mantenibilidad, la capacidad de prueba, la deuda técnica y las arquitecturas monolítica y de microservicios. Tras la selección, los materiales se analizaron y compararon con el fin de identificar los principales argumentos favorables y contrarios a la adopción de la Clean Architecture, así como los contextos en los que su utilización puede contribuir o dificultar el desarrollo de software.

2 Desarrollo

2.1 Arquitectura de Software y Desarrollo de Sistemas Modernos

La arquitectura de software corresponde a la estructura que organiza los componentes de un sistema, define sus responsabilidades y establece las formas de comunicación entre ellos. Incluye decisiones sobre tecnologías, patrones, módulos e integraciones, e influye directamente en el funcionamiento, el mantenimiento y la capacidad de evolución de la aplicación a lo largo de su ciclo de vida (Ciriano; Pinto, 2025).

En los sistemas modernos, las decisiones arquitectónicas se han vuelto más importantes debido al aumento de la complejidad. Las aplicaciones necesitan atender a diferentes usuarios, integrar servicios externos, procesar información y funcionar en diversas plataformas. Una estructura poco organizada puede dificultar la incorporación de nuevas funcionalidades y aumentar los riesgos de fallos durante los cambios.

La calidad del software también depende de factores como la mantenibilidad, la fiabilidad, la flexibilidad y la capacidad de prueba. Incluso cuando cumple los requisitos iniciales, una aplicación puede volverse difícil de modificar si presenta dependencias excesivas y responsabilidades mal distribuidas. La arquitectura contribuye a reducir estos problemas al establecer límites entre los componentes (Escareli; Malara, 2022).

El mantenimiento incluye correcciones, mejoras, actualizaciones y adaptaciones a nuevas necesidades. Cuando no existe una separación adecuada de responsabilidades, pequeños cambios pueden afectar a diferentes módulos. Por ello, la planificación arquitectónica debe considerar no solo la entrega inicial, sino también la evolución futura del sistema.

Los cambios frecuentes en frameworks, bibliotecas y bases de datos también influyen en la arquitectura. Cuando las reglas principales están directamente vinculadas a estas tecnologías, su sustitución puede exigir modificaciones extensas. Los componentes reutilizables y los mecanismos de integración bien definidos reducen estas dependencias y favorecen sistemas más flexibles (Gomes; Gibertoni, 2022).

La separación entre las reglas de negocio y los detalles tecnológicos es fundamental. Las reglas expresan los procesos centrales de la aplicación, mientras que las interfaces, los servidores y los mecanismos de persistencia funcionan como recursos de apoyo. Mezclar estas dimensiones hace que los cambios externos afecten directamente al funcionamiento principal del software.

También debe controlarse el acoplamiento entre los componentes. Cuando los módulos poseen dependencias intensas, los cambios aislados pueden exigir modificaciones en varias partes del sistema. La inversión de dependencias reduce este problema al orientar a los componentes centrales para que dependan de abstracciones y no de implementaciones específicas (Gonçalves; Mendes, 2022).

Además del bajo acoplamiento, la alta cohesión favorece la organización del código. Un componente cohesivo reúne funciones relacionadas y posee una responsabilidad bien definida. Esto facilita la localización de las reglas, la comprensión del sistema y la realización de cambios con mayor seguridad.

La arquitectura también influye en la capacidad de prueba. Los componentes independientes pueden verificarse de forma aislada, reduciendo la necesidad de ejecutar toda la aplicación. Las pruebas unitarias, de integración y de interfaz ayudan a identificar fallos de manera temprana y aumentan la confianza durante las modificaciones (Lima; Leal; Mesquita, 2023).

Sin embargo, las pruebas no corrigen los problemas provocados por una arquitectura inadecuada. Cuando los módulos permanecen fuertemente vinculados, su validación puede exigir muchas configuraciones y recursos externos. Así, la calidad de las pruebas también depende de la forma en que se hayan organizado las responsabilidades y las dependencias.

La falta de planificación puede contribuir a la aparición de deuda técnica. Los atajos, las duplicaciones, las dependencias inadecuadas y la ausencia de documentación aceleran las

entregas a corto plazo, pero hacen que los cambios futuros sean más caros y complejos. En sistemas heredados, la acumulación de estas decisiones puede limitar la evolución de la aplicación (Serafim; Duarte; Duarte, 2022).

Por otra parte, una arquitectura excesivamente elaborada también puede causar dificultades. Los sistemas simples pueden no necesitar muchas capas, interfaces o abstracciones. En estos casos, la estructura puede aumentar el volumen de código, prolongar el desarrollo y dificultar la comprensión por parte del equipo.

Las arquitecturas monolíticas y de microservicios presentan ventajas y limitaciones. El monolito reúne las funcionalidades en una sola aplicación, mientras que los microservicios distribuyen responsabilidades entre servicios independientes. La elección debe considerar la complejidad del negocio, el tamaño del equipo, la escalabilidad y la capacidad para administrar la infraestructura (Pegrucci; Tiosso; Reis, 2020).

Ningún enfoque elimina todos los problemas. Una arquitectura distribuida puede favorecer la escalabilidad, pero crear dificultades de comunicación e integración. Del mismo modo, muchas capas pueden mejorar la separación de responsabilidades, aunque aumenten el esfuerzo inicial. Por ello, las decisiones deben considerar los beneficios y los costes de cada alternativa.

En este contexto, la Clean Architecture busca organizar las dependencias y proteger las reglas centrales del sistema. Al mantener los frameworks, las bases de datos y las interfaces en las capas externas, procura facilitar las pruebas, el mantenimiento y las sustituciones tecnológicas. No obstante, la creación de límites y abstracciones también puede aumentar la complejidad inicial (Donato; Zuchi, 2021).

Por lo tanto, la arquitectura debe utilizarse como un medio para atender las necesidades del proyecto. La elección debe considerar el contexto, los recursos disponibles, la experiencia del equipo y la expectativa de evolución. Una estructura adecuada favorece la calidad y el mantenimiento, mientras que una solución desproporcionada puede generar costes y dificultades innecesarias.

2.2 Origen y Fundamentos de la Clean Architecture

La Clean Architecture surgió como una propuesta para reducir las dependencias entre las reglas de negocio y los elementos tecnológicos de la aplicación. Su estructura establece límites entre las responsabilidades del sistema, protegiendo el núcleo frente a cambios en frameworks, bases de datos, interfaces y servicios externos. Con ello, busca facilitar el

mantenimiento y la adaptación del software a lo largo del tiempo (Ferreira; Ferreira; Grande, 2022).

Su origen está relacionado con modelos como la arquitectura en capas, la arquitectura hexagonal y la Onion Architecture. A pesar de las diferencias entre estos enfoques, todos defienden que las reglas centrales no deben depender directamente de los detalles responsables de la presentación, la persistencia o la comunicación con sistemas externos.

En la arquitectura en capas, las responsabilidades suelen distribuirse entre presentación, aplicación, dominio e infraestructura. Esta división facilita la identificación de las funciones de cada componente, siempre que las dependencias se planifiquen correctamente (Ciriano; Pinto, 2025).

Sin embargo, limitarse a dividir el software en capas no garantiza la independencia del núcleo. Cuando las reglas de negocio continúan dependiendo de recursos específicos de la infraestructura, los cambios tecnológicos pueden afectar directamente a la lógica de la aplicación.

La arquitectura hexagonal, o Ports and Adapters, propone aislar el núcleo de los mecanismos de entrada y salida. Los puertos definen contratos de comunicación, mientras que los adaptadores conectan el sistema con interfaces, bases de datos y servicios externos (Silva; Costa, 2024).

La Onion Architecture también organiza el sistema en círculos concéntricos. Las entidades y las reglas de dominio permanecen en el centro, mientras que los servicios, las interfaces y la infraestructura se sitúan en las capas externas. Las dependencias siempre deben dirigirse hacia el interior.

La Clean Architecture reúne elementos de estas propuestas y determina que las capas internas no conozcan detalles externos. De este modo, las entidades y los casos de uso permanecen independientes de la base de datos, el framework y la interfaz utilizada por el usuario (Donato; Zuchi, 2021).

Esta protección se produce porque las reglas de negocio representan la parte más importante y estable del sistema. Las herramientas y tecnologías pueden sustituirse, mientras que el dominio expresa los procesos y comportamientos que justifican la existencia de la aplicación.

La inversión de dependencias refuerza este fundamento. En lugar de que el núcleo dependa directamente de clases concretas de la infraestructura, utiliza abstracciones y contratos. Las capas externas pasan a implementar las necesidades definidas por las partes internas, reduciendo el acoplamiento (Gonçalves; Mendes, 2022).

La inversión de dependencias no se limita al uso de herramientas de inyección. Se trata de reorganizar la relación entre los módulos, haciendo que los componentes más importantes no dependan de aquellos sujetos a cambios frecuentes.

La capacidad de prueba también constituye un fundamento de la Clean Architecture. Cuando las reglas de negocio están separadas de bases de datos, interfaces y servicios externos, pueden verificarse de manera aislada. Las dependencias reales pueden sustituirse por implementaciones simuladas, haciendo que las pruebas sean más rápidas y específicas (Lima; Leal; Mesquita, 2023).

Esta facilidad aumenta la seguridad durante los cambios, pues las pruebas ayudan a confirmar que los comportamientos anteriores se han preservado. Sin embargo, una estructura comprobable debe planificarse, ya que separar posteriormente componentes fuertemente acoplados puede exigir un gran esfuerzo.

La independencia respecto de los frameworks también es relevante. Estas herramientas deben tratarse como recursos utilizados por la aplicación y no como elementos responsables de controlar su lógica central. Mantener las configuraciones específicas en las capas externas reduce los impactos de futuras sustituciones tecnológicas (Ferreira; Ferreira; Grande, 2022).

Esto no significa rechazar frameworks y bibliotecas. Estos recursos aceleran el desarrollo y ofrecen soluciones útiles. El objetivo es impedir que sus características determinen por completo la estructura de las reglas de negocio.

El mismo principio se aplica a las bases de datos. El acceso a la información debe realizarse mediante contratos, evitando que las entidades se modelen exclusivamente según tablas, documentos o recursos de una tecnología específica de persistencia (Gomes; Gibertoni, 2022).

Esta separación puede reducir los impactos de sustituir el mecanismo de almacenamiento o integrar nuevas fuentes de datos. Sin embargo, no garantiza sustituciones automáticas, pues cada tecnología posee características propias.

La independencia de la interfaz permite que los mismos casos de uso sean accesibles desde aplicaciones web, sistemas móviles, interfaces de línea de comandos o integraciones externas. De este modo, la lógica central puede reutilizarse sin duplicación entre diferentes plataformas (Nascimento; Sotto, 2020).

A pesar de ello, la Clean Architecture no debe entenderse como un modelo rígido de carpetas o nombres. Su aplicación puede variar según el lenguaje, el tamaño del sistema y la experiencia del equipo. Lo esencial es preservar la dirección de las dependencias y separar las reglas de negocio de los detalles tecnológicos.

Una adopción excesivamente formal puede generar interfaces, conversores y clases sin utilidad concreta. Estos elementos deben crearse únicamente cuando ayuden a organizar responsabilidades, reducir dependencias o facilitar pruebas, evitando una complejidad innecesaria (Donato; Zuchi, 2021).

Por lo tanto, la Clean Architecture procura proteger el núcleo, invertir las dependencias y mantener las tecnologías en las capas externas. Estos fundamentos pueden favorecer el mantenimiento, la capacidad de prueba y la adaptación, siempre que se apliquen de forma proporcional a las necesidades del proyecto.

2.3 Principios y Organización de las Capas de la Clean Architecture

La Clean Architecture organiza el sistema en capas con responsabilidades distintas, manteniendo las reglas de negocio protegidas de los detalles tecnológicos. En su representación tradicional, el núcleo concentra los elementos más importantes, mientras que las interfaces, bases de datos, frameworks y servicios externos permanecen en las capas periféricas. Esta estructura busca reducir el impacto de los cambios tecnológicos sobre la lógica central de la aplicación (Ferreira; Ferreira; Grande, 2022).

La implementación no exige una cantidad fija de capas. Su organización puede variar según el lenguaje, la complejidad del proyecto y las necesidades del equipo. El aspecto principal consiste en separar correctamente las responsabilidades y garantizar que los componentes internos no dependan directamente de las tecnologías externas.

Las entidades ocupan la capa más interna y representan las reglas generales y estables del dominio. Pueden reunir datos, comportamientos y restricciones relacionados con el funcionamiento del sistema, permaneciendo independientes de bases de datos, interfaces gráficas y bibliotecas específicas (Donato; Zuchi, 2021).

Estas entidades no deben funcionar únicamente como estructuras de almacenamiento. Deben expresar conceptos relevantes y concentrar comportamientos del dominio. Cuando las reglas se dispersan entre controladores y servicios externos, el sistema pierde cohesión y se vuelve más difícil de comprender.

La capa de casos de uso representa las acciones ofrecidas por la aplicación. Estos componentes coordinan el flujo de información, utilizan las entidades y ejecutan reglas específicas. De este modo, describen lo que realiza el sistema sin depender directamente de la forma de presentación o almacenamiento de los datos (Silva; Costa, 2024).

Cada caso de uso debe poseer una responsabilidad bien definida. Al recibir una solicitud, valida los datos, interactúa con las entidades y utiliza contratos para acceder a recursos externos. Esta organización facilita las pruebas y permite que diferentes interfaces compartan las mismas funcionalidades.

Los adaptadores de interfaz transforman los datos entre los formatos internos y externos. En esta capa pueden encontrarse controladores, presentadores, conversores, gateways e implementaciones de repositorios. Su función es permitir la comunicación con el núcleo sin exponer las reglas de negocio a las particularidades tecnológicas (Gomes; Gibertoni, 2022).

Los controladores envían las solicitudes a los casos de uso, mientras que los presentadores organizan las respuestas. Estos componentes no deben concentrar reglas de negocio, pues la acumulación de responsabilidades debilita la separación propuesta por la arquitectura.

En la capa externa se encuentran frameworks, bases de datos, servidores, interfaces y servicios de terceros. Estos elementos se consideran detalles de implementación y deben proporcionar recursos tecnológicos sin controlar las reglas del dominio (Ciriano; Pinto, 2025).

Las configuraciones de inicio, conexión, rutas y gestión de dependencias también permanecen en esta capa. Como estos recursos cambian con mayor frecuencia, mantenerlos alejados del núcleo reduce la propagación de los impactos tecnológicos por el sistema.

La regla de dependencia determina que las relaciones del código apunten hacia las capas internas. Las entidades no conocen los casos de uso y los casos de uso no acceden directamente a la infraestructura. Las capas externas implementan los contratos definidos por el núcleo, preservando la independencia de las reglas de negocio (Gonçalves; Mendes, 2022).

Cuando un caso de uso necesita almacenar información, por ejemplo, utiliza una interfaz en lugar de acceder directamente a una clase de base de datos. La implementación concreta permanece en la infraestructura y se proporciona durante la ejecución, de modo que el núcleo solo conoce el contrato.

Este proceso está relacionado con la inversión de dependencias. El núcleo define aquello que necesita, mientras que las capas externas se adaptan a esas exigencias. La inyección de dependencias puede utilizarse para proporcionar implementaciones reales o simuladas, favoreciendo la sustitución de componentes durante las pruebas (Lima; Leal; Mesquita, 2023).

La inversión de dependencias es un principio arquitectónico, mientras que la inyección corresponde a una técnica. El simple uso de una herramienta de inyección no garantiza un bajo acoplamiento, especialmente cuando los módulos internos siguen conociendo clases concretas de la infraestructura.

Los principios SOLID también contribuyen a esta organización. Orientan la definición de responsabilidades, la creación de contratos específicos, la posibilidad de extensión y la dependencia de abstracciones. Cuando se aplican de manera equilibrada, favorecen componentes más cohesivos y fáciles de probar (Ferreira; Ferreira; Grande, 2022).

Sin embargo, estos principios no deben provocar una fragmentación excesiva. Crear interfaces y clases para operaciones simples puede aumentar el número de archivos sin generar una independencia relevante. Cada abstracción debe tener una finalidad concreta, como facilitar pruebas, permitir sustituciones o aislar tecnologías.

El flujo de datos atraviesa las capas mediante contratos definidos. Una solicitud parte de la interfaz, pasa por el controlador y llega al caso de uso, que ejecuta las reglas e interactúa con las entidades. La respuesta recorre el camino inverso y se adapta al formato esperado por el usuario (Donato; Zuchi, 2021).

Pueden utilizarse objetos propios para transportar datos entre las capas y evitar que las entidades dependan de frameworks. Sin embargo, el exceso de modelos y conversores puede generar código repetitivo. La decisión debe equilibrar independencia y simplicidad.

El bajo acoplamiento y la alta cohesión son resultados esperados de esta organización. El primero limita el conocimiento entre componentes, mientras que la segunda concentra actividades relacionadas. Esto reduce los efectos inesperados de los cambios y facilita la localización de las reglas en el sistema (Escareli; Malara, 2022).

No obstante, la división en capas no garantiza automáticamente una buena arquitectura. Un proyecto puede tener muchas carpetas e interfaces y aun así presentar reglas duplicadas o responsabilidades inadecuadas. La calidad depende de la definición y preservación de los límites durante el desarrollo.

La Clean Architecture puede emplearse tanto en aplicaciones monolíticas como en microservicios. En el monolito, organiza internamente los módulos; en los microservicios, puede estructurar cada servicio de manera independiente. Por lo tanto, estos enfoques actúan en niveles diferentes y no son necesariamente competidores (Kanizawa; Pinto, 2022).

En los sistemas distribuidos, sin embargo, siguen existiendo dificultades como fallos de red, consistencia de datos y monitorización. La Clean Architecture mejora la organización interna de los servicios, pero no elimina la complejidad operativa de este modelo.

La separación de las capas también favorece la capacidad de prueba. Los casos de uso y las entidades pueden verificarse sin ejecutar bases de datos o interfaces reales, utilizando implementaciones simuladas. Aun así, las pruebas de integración continúan siendo necesarias para validar la comunicación entre los componentes (Romanini; Sotto, 2019).

Por lo tanto, la Clean Architecture busca proteger las reglas de negocio y controlar las dependencias tecnológicas mediante entidades, casos de uso, adaptadores y componentes externos. Cuando se aplica de forma proporcional, puede mejorar la claridad, el mantenimiento y las pruebas. Sin embargo, debe evitar divisiones y abstracciones que no presenten una utilidad concreta para el proyecto.

2.4 La Clean Architecture como Aliada del Desarrollo de Software

La Clean Architecture puede actuar como aliada del desarrollo de software al organizar el código y reducir las dependencias entre los componentes. Al separar las reglas de negocio de frameworks, bases de datos, interfaces y servicios externos, protege el núcleo de la aplicación frente a cambios tecnológicos y facilita su mantenimiento y evolución (Donato; Zuchi, 2021).

La definición clara de las responsabilidades también favorece la comprensión del sistema. Las entidades concentran reglas del dominio, los casos de uso coordinan funcionalidades, los adaptadores convierten datos y los componentes externos proporcionan recursos tecnológicos. Esta división reduce la dispersión de las reglas y facilita la identificación de los puntos que deben modificarse.

La mantenibilidad es una de las principales ventajas del enfoque. En sistemas poco organizados, las clases extensas, las responsabilidades mezcladas y las dependencias excesivas hacen que pequeños cambios afecten a diferentes módulos. La separación en capas limita estos impactos y mantiene las modificaciones cerca de los componentes implicados (Escareli; Malara, 2022).

Esta organización también facilita la incorporación de funcionalidades, la adaptación a nuevas reglas y la actualización de tecnologías. Al considerar la evolución desde el principio, el equipo reduce el esfuerzo necesario para comprender el código y realizar modificaciones futuras.

La independencia respecto de los frameworks contribuye a preservar las reglas de negocio cuando una tecnología necesita actualizarse o sustituirse. Aunque el cambio todavía exige esfuerzo, mantener los detalles tecnológicos en las capas externas reduce la cantidad de componentes afectados (Ferreira; Ferreira; Grande, 2022).

La capacidad de prueba constituye otro beneficio relevante. Cuando las entidades y los casos de uso no dependen directamente de bases de datos, redes o interfaces, sus comportamientos pueden verificarse de forma aislada con dependencias simuladas. Esto favorece pruebas más rápidas y facilita la identificación de fallos.

La combinación de pruebas unitarias, de integración y de interfaz amplía la calidad del sistema. Sin embargo, la existencia de capas no garantiza por sí sola una aplicación comprobable. Los componentes deben poseer responsabilidades bien definidas y respetar la dirección de las dependencias (Lima; Leal; Mesquita, 2023).

La flexibilidad tecnológica también puede verse favorecida por el uso de abstracciones. Las bases de datos, los servicios externos y otros recursos pueden sustituirse con menor impacto sobre el núcleo, siempre que el acceso se realice mediante contratos bien definidos.

Esta característica es útil en sistemas utilizados por diferentes plataformas. Las aplicaciones web y móviles y los servicios de integración pueden compartir los mismos casos de uso, reduciendo duplicaciones y garantizando una mayor consistencia de las reglas de negocio (Gonçalves; Mendes, 2022).

La reutilización también ocurre cuando las funcionalidades comunes permanecen concentradas en el núcleo. En lugar de repetir validaciones y procesos en diferentes interfaces, los casos de uso proporcionan un único punto de ejecución. Sin embargo, los componentes no deben transformarse en estructuras excesivamente genéricas solo para aumentar la reutilización.

En sistemas más grandes, los límites entre las capas pueden facilitar la división del trabajo entre equipos. Diferentes grupos pueden trabajar en el dominio, las interfaces o la infraestructura con menor interferencia entre sus actividades. Una estructura previsible también favorece las revisiones de código y la incorporación de nuevos integrantes (Ciriano; Pinto, 2025).

La Clean Architecture puede aplicarse tanto en monolitos como en microservicios. En sistemas monolíticos, organiza internamente los módulos; en los microservicios, separa las reglas de cada servicio de los mecanismos de comunicación y persistencia. Sin embargo, no resuelve problemas propios de los sistemas distribuidos, como los fallos de red y la consistencia de los datos.

La separación de responsabilidades también puede reducir la deuda técnica. Cuando las reglas, interfaces y detalles tecnológicos están mezclados, cada cambio tiende a aumentar la complejidad. Los límites bien definidos facilitan las refactorizaciones e impiden que las soluciones temporales se propaguen por el código (Serafim; Duarte; Duarte, 2022).

A pesar de ello, la arquitectura no elimina los problemas causados por la falta de pruebas, documentación o planificación. Sus beneficios dependen de prácticas consistentes y del respeto a los límites definidos durante el desarrollo.

La longevidad del software también puede verse favorecida. Los sistemas corporativos permanecen en uso durante muchos años y atraviesan cambios tecnológicos. Cuando las reglas

centrales están aisladas, las interfaces, herramientas y mecanismos de almacenamiento pueden adaptarse sin exigir la reescritura completa de la aplicación (Silva; Saito, 2021).

Los principios SOLID refuerzan esta organización al favorecer la responsabilidad única, la segregación de interfaces y la inversión de dependencias. Sin embargo, las interfaces y abstracciones deben utilizarse únicamente cuando solucionan problemas concretos, evitando el aumento innecesario del número de clases y archivos.

La previsibilidad de la estructura también mejora la comunicación entre los desarrolladores. Cuando el equipo comprende la función de cada capa, resulta más fácil definir dónde incluir nuevas reglas, qué componentes pueden acceder a servicios externos y qué dependencias están permitidas (Silva; Costa, 2024).

Así, la Clean Architecture actúa como aliada principalmente en proyectos con reglas complejas, larga vida útil, múltiples integraciones y necesidad frecuente de evolución. En estos contextos, el esfuerzo inicial puede compensarse con una mayor facilidad de mantenimiento, prueba y adaptación tecnológica.

En síntesis, el enfoque puede favorecer la mantenibilidad, la capacidad de prueba, la flexibilidad, la reutilización y la longevidad. Sin embargo, sus beneficios dependen de una aplicación proporcional a las características del proyecto. Cuando se utiliza de forma rígida o sin una necesidad concreta, puede comprometer la simplicidad del desarrollo (Donato; Zuchi, 2021).

2.5 Limitaciones y Desafíos de la Implementación de la Clean Architecture

Aunque la Clean Architecture favorece la organización, el mantenimiento y la capacidad de prueba, su implementación también presenta limitaciones. La creación de capas, contratos, interfaces y adaptadores puede ayudar a controlar dependencias en sistemas complejos, pero, cuando se aplica sin necesidad, aumenta el volumen de código y hace que el desarrollo sea más laborioso (Donato; Zuchi, 2021).

Uno de los principales desafíos es la complejidad inicial. El equipo necesita definir con antelación qué reglas pertenecen al dominio, qué operaciones se tratarán como casos de uso y qué recursos permanecerán en las capas externas. Esta planificación exige más tiempo que las estructuras basadas directamente en los recursos ofrecidos por los frameworks.

La división de una funcionalidad entre entidades, controladores, casos de uso, repositorios y modelos de datos puede dificultar la comprensión del flujo. Aunque cada componente tiene una responsabilidad específica, la necesidad de consultar muchos archivos

puede hacer que la lectura sea menos directa, especialmente para los desarrolladores poco familiarizados con el enfoque (Ferreira; Ferreira; Grande, 2022).

La curva de aprendizaje también representa un obstáculo. Conceptos como inversión de dependencias, dominio, abstracciones y adaptadores deben comprenderse correctamente. De lo contrario, el proyecto puede limitarse a reproducir una división de carpetas, manteniendo las reglas de negocio dependientes de frameworks y recursos de infraestructura.

Esta implementación superficial aumenta la cantidad de componentes sin alcanzar la independencia esperada. Por ello, la adopción de la Clean Architecture exige capacitación, documentación y revisiones de código, además de mayores conocimientos técnicos por parte de los desarrolladores (Gonçalves; Mendes, 2022).

Los costes iniciales también pueden aumentar, sobre todo en equipos inexpertos. La formación, el seguimiento y las decisiones equivocadas sobre los límites entre las capas pueden prolongar el desarrollo y exigir refactorizaciones posteriores. Sus beneficios suelen aparecer principalmente durante el mantenimiento y la evolución del sistema.

En proyectos con plazos reducidos, la creación de contratos y abstracciones puede retrasar la entrega de funcionalidades simples. El equipo debe equilibrar la velocidad de desarrollo con la necesidad de evitar una estructura difícil de mantener en el futuro (Ciriano; Pinto, 2025).

Otro riesgo es la creación de abstracciones innecesarias. Las interfaces y los adaptadores son útiles cuando protegen el núcleo de dependencias relevantes, facilitan pruebas o permiten sustituciones. Sin embargo, crear una interfaz para cada clase puede aumentar la complejidad sin ofrecer beneficios proporcionales.

Las abstracciones deben resolver problemas concretos. Cuando una implementación es simple, estable y tiene pocas posibilidades de ser sustituida, el acceso directo puede ser más adecuado. El uso excesivo de intermediarios dificulta la lectura del código y aumenta el esfuerzo de mantenimiento (Escareli; Malara, 2022).

La duplicación de modelos también puede elevar el volumen de código. La misma información puede estar representada por entidades, modelos de persistencia, objetos de entrada y estructuras de respuesta. Aunque preserva la independencia de las capas, esta separación exige conversiones constantes entre formatos.

En sistemas con muchos campos y operaciones simples, cualquier cambio puede exigir modificaciones en varios modelos y conversores. El equipo debe evaluar si el aislamiento obtenido compensa el esfuerzo o si determinadas estructuras pueden compartirse sin comprometer las reglas de negocio (Silva; Costa, 2024).

En proyectos pequeños, prototipos o aplicaciones de corta duración, la adopción integral de la Clean Architecture puede ser desproporcionada. La creación de diversas capas puede aumentar el tiempo de desarrollo sin ofrecer beneficios relevantes de mantenimiento o flexibilidad.

En estos contextos, una estructura simplificada puede proporcionar mayor agilidad. Si el producto crece, pueden introducirse nuevos límites gradualmente. Sin embargo, esta estrategia debe evitar que las soluciones temporales se acumulen y formen una deuda técnica difícil de controlar (Donato; Zuchi, 2021).

El exceso de planificación también puede llevar a la creación de estructuras destinadas a cambios futuros que quizá nunca ocurran. Las interfaces, adaptadores y puntos de extensión basados únicamente en posibilidades remotas aumentan los costes actuales y pueden hacer que el sistema sea más difícil de comprender.

Por otra parte, la falta de planificación favorece el alto acoplamiento y la deuda técnica. El desafío consiste en preparar el sistema para cambios probables sin crear componentes destinados únicamente a escenarios hipotéticos (Serafim; Duarte; Duarte, 2022).

La integración con frameworks representa otra dificultad. Muchas herramientas ofrecen estructuras propias para controladores, entidades y acceso a datos. El aislamiento completo del núcleo puede exigir adaptadores adicionales e impedir el aprovechamiento de algunas facilidades proporcionadas por la tecnología utilizada.

La independencia tecnológica no significa rechazar frameworks. Estos recursos pueden acelerar el desarrollo y estandarizar tareas. En determinados proyectos, aceptar una dependencia limitada y conocida puede ser más adecuado que crear una estructura extensa solo para una posible sustitución futura (Ferreira; Ferreira; Grande, 2022).

Las pruebas también exigen equilibrio. Aunque la Clean Architecture facilita las pruebas aisladas, el uso excesivo de objetos simulados puede producir resultados alejados del comportamiento real. Una funcionalidad puede superar las pruebas unitarias y fallar durante la integración con bases de datos, redes o servicios externos.

Por ello, las pruebas unitarias deben combinarse con pruebas de integración y de interfaz. Además, los cambios en los contratos pueden exigir la actualización de muchos objetos simulados, aumentando el trabajo de mantenimiento de las propias pruebas (Lima; Leal; Mesquita, 2023).

En sistemas distribuidos, la Clean Architecture no elimina los problemas de los microservicios. La comunicación por red, la falta de disponibilidad, la seguridad, la consistencia

de los datos y la monitorización continúan siendo desafíos. La combinación de ambos enfoques puede elevar considerablemente la cantidad de componentes administrados por el equipo.

En equipos pequeños, esta complejidad puede superar los beneficios esperados. Cuando no existen requisitos concretos de distribución o escalabilidad independiente, un monolito organizado puede representar una solución más simple y eficiente (Kanizawa; Pinto, 2022).

La consistencia arquitectónica también debe mantenerse a lo largo del tiempo. Los plazos cortos, los cambios en el equipo y las entregas urgentes pueden llevar a incluir reglas de negocio en los controladores o al acceso directo de la infraestructura por parte de los casos de uso.

Para evitar esta degradación, son necesarias revisiones de código, documentación, estándares internos y seguimiento continuo. La arquitectura no permanece organizada únicamente por las decisiones tomadas al inicio del proyecto (Escareli; Malara, 2022).

La aplicación rígida de los principios también puede impedir soluciones simples y adecuadas. Las interfaces y los intermediarios no deben crearse solo para reproducir un modelo teórico. La arquitectura debe orientar las decisiones, pero permitir adaptaciones según las características reales del sistema.

El pragmatismo exige comparar los costes y beneficios de cada separación. Una dependencia puede ser aceptable cuando es limitada, conocida y poco susceptible de cambios, mientras que una abstracción se justifica cuando protege reglas importantes o favorece pruebas fiables (Gomes; Gibertoni, 2022).

Por lo tanto, las limitaciones de la Clean Architecture no hacen que el enfoque sea inadecuado, sino que demuestran que su adopción depende del contexto. Deben considerarse la complejidad inicial, el aumento del código, la curva de aprendizaje, el exceso de abstracciones y los costes de mantenimiento.

En sistemas duraderos y con reglas complejas, la inversión arquitectónica puede compensarse a lo largo del tiempo. En aplicaciones simples, una adopción parcial puede ser más adecuada. Así, la Clean Architecture se convierte en una posible villana cuando se aplica indiscriminadamente y sin relación con las necesidades concretas del proyecto (Donato; Zuchi, 2021).

2.6 Adecuación de la Clean Architecture a Diferentes Contextos de Desarrollo

La adopción de la Clean Architecture debe considerar las características de cada proyecto, como el tamaño de la aplicación, la complejidad de las reglas, la experiencia del

equipo y la expectativa de evolución. Aunque sus principios favorecen la separación de responsabilidades y el control de las dependencias, su aplicación integral no siempre es necesaria. La arquitectura debe ser proporcional a las necesidades del software, evitando tanto la desorganización como la creación de estructuras excesivamente complejas (Donato; Zuchi, 2021).

En sistemas con reglas de negocio complejas y actualizaciones frecuentes, la separación entre dominio, casos de uso, interfaces e infraestructura puede facilitar la comprensión, las pruebas y el mantenimiento. Al mantener las reglas centrales en componentes independientes, los cambios tienden a producir un menor impacto sobre las demás partes de la aplicación.

Los proyectos corporativos de larga duración pueden beneficiarse de esta organización, pues generalmente pasan por diferentes equipos, cambios en las reglas de negocio y sustituciones tecnológicas. La preservación del núcleo de la aplicación reduce los impactos provocados por el cambio de frameworks, bases de datos e interfaces, contribuyendo a la continuidad del sistema (Ferreira; Ferreira; Grande, 2022).

Sin embargo, no toda aplicación corporativa exige muchas capas y abstracciones. Los sistemas administrativos con flujos simples y pocos cambios pueden organizarse mediante estructuras menores. La elección debe basarse en la complejidad de las funcionalidades y en los costes de mantenimiento, y no únicamente en el tamaño de la organización.

En proyectos pequeños, la aplicación integral de la Clean Architecture puede ser desproporcionada. La creación de interfaces, adaptadores y modelos para cada capa puede aumentar el esfuerzo sin proporcionar beneficios significativos. En estos casos, una estructura más simple puede preservar la calidad y facilitar el desarrollo (Escareli; Malara, 2022).

Esto no significa abandonar sus principios. La separación entre las reglas de negocio y la presentación, la definición de responsabilidades y la reducción del acoplamiento pueden aplicarse de manera simplificada. El equipo puede adoptar solo los elementos necesarios, evitando una estructura mayor de lo que exige el problema.

Las startups y los productos en validación también requieren equilibrio. Una arquitectura extensa puede retrasar la entrega de la primera versión, mientras que la ausencia de organización puede dificultar el crecimiento futuro. Así, es posible comenzar con una estructura simple e introducir nuevos límites a medida que los requisitos se vuelven más claros (Serafim; Duarte; Duarte, 2022).

La experiencia del equipo influye directamente en los resultados. Los desarrolladores familiarizados con los principios SOLID y la inversión de dependencias tienden a aplicar el enfoque con mayor seguridad. En equipos principiantes, la falta de comprensión puede generar

componentes fragmentados y abstracciones sin finalidad. Por ello, la documentación, la capacitación y las revisiones de código son fundamentales.

La Clean Architecture también puede aplicarse en sistemas monolíticos. Una aplicación implementada como una sola unidad puede poseer módulos bien definidos, casos de uso independientes y separación entre dominio e infraestructura. En muchos proyectos, un monolito organizado resulta más adecuado que la adopción inmediata de microservicios (Pegrucci; Tiosso; Reis, 2020).

En los microservicios, sus principios pueden organizar internamente cada servicio, protegiendo las reglas de negocio de los mecanismos de comunicación y persistencia. Sin embargo, esta combinación aumenta la cantidad de componentes y no elimina desafíos como los fallos de red, la consistencia de los datos, la monitorización y la complejidad de la implementación.

En sistemas heredados, la Clean Architecture puede orientar una modernización gradual. La creación progresiva de límites, casos de uso y adaptadores permite aislar partes del código antiguo y reducir dependencias sin exigir la reescritura completa de la aplicación, disminuyendo los riesgos de la transformación (Silva; Saito, 2021).

La integración con diferentes interfaces también puede justificar su adopción. Las aplicaciones web y móviles y los servicios externos pueden compartir los mismos casos de uso cuando la lógica central permanece independiente de la presentación. Sin embargo, en sistemas con una sola interfaz simple y estable, una separación menos extensa puede ser suficiente.

Los requisitos de capacidad de prueba constituyen otro criterio importante. En sistemas en los que los fallos pueden generar pérdidas relevantes, el aislamiento de las reglas facilita la realización de pruebas unitarias y la sustitución de dependencias externas por implementaciones controladas. Esta característica contribuye a validar los comportamientos centrales con mayor seguridad (Lima; Leal; Mesquita, 2023).

También debe observarse la frecuencia de los cambios tecnológicos. Los contratos y adaptadores son más útiles cuando existe dependencia de servicios externos, diferentes fuentes de datos o tecnologías sujetas a sustitución. Cuando los recursos utilizados son estables, el coste de crear y mantener abstracciones puede superar los beneficios esperados.

El tiempo y los recursos disponibles influyen igualmente en la decisión. La Clean Architecture suele exigir una mayor inversión inicial en la definición de capas, contratos y flujos de datos. En proyectos duraderos, este esfuerzo puede compensarse con la facilidad de mantenimiento, pero en sistemas temporales quizá no se obtenga el retorno (Ciriano; Pinto, 2025).

Por lo tanto, no existe una respuesta única para definir la Clean Architecture como aliada o villana. Puede facilitar el desarrollo de sistemas complejos y dificultar innecesariamente la construcción de aplicaciones simples. Su adopción debe considerar la complejidad del dominio, el número de integraciones, el crecimiento esperado, la duración del proyecto, la experiencia del equipo y la necesidad de pruebas.

El enfoque también puede aplicarse de forma parcial y progresiva. El sistema puede comenzar con la separación entre dominio e infraestructura e incorporar nuevos casos de uso, adaptadores y abstracciones a medida que surjan necesidades concretas. De esta manera, la arquitectura acompaña la evolución del software sin tratarse como un modelo rígido (Donato; Zuchi, 2021).

Así, la Clean Architecture actúa como aliada cuando protege reglas relevantes, facilita pruebas, reduce dependencias y favorece la evolución. Se convierte en una posible villana cuando se aplica indiscriminadamente, aumentando el volumen de código, los costes y la dificultad de comprensión. Su efectividad depende de una utilización crítica, pragmática y coherente con las condiciones de cada proyecto.

3 Conclusión

La realización de este trabajo permitió analizar críticamente la aplicación de la Clean Architecture en el desarrollo de software moderno, considerando sus fundamentos, beneficios, limitaciones y diferentes contextos de utilización. A partir de la revisión bibliográfica, fue posible comprender que este enfoque busca proteger las reglas de negocio de los cambios tecnológicos, organizando el sistema mediante la separación de responsabilidades y el control de las dependencias entre los componentes.

Entre los principales beneficios identificados se encuentran la mejora de la mantenibilidad, la capacidad de prueba, la flexibilidad y la capacidad de evolución de los sistemas. Al mantener el núcleo de la aplicación independiente de frameworks, bases de datos, interfaces y servicios externos, la Clean Architecture puede reducir los impactos provocados por cambios tecnológicos y facilitar la realización de pruebas aisladas. Además, la definición de límites claros entre entidades, casos de uso, adaptadores e infraestructura contribuye a una organización más comprensible del código.

Sin embargo, también se identificaron limitaciones importantes. La implementación de la Clean Architecture puede aumentar la cantidad de clases, interfaces, contratos, conversores y archivos, haciendo que el desarrollo sea inicialmente más lento y complejo. Este problema es

más evidente en proyectos pequeños, prototipos, aplicaciones temporales o sistemas con reglas de negocio simples, en los que el esfuerzo de crear una estructura extensa puede no verse compensado por beneficios futuros.

Otro desafío está relacionado con la experiencia del equipo. La adopción adecuada del enfoque exige conocimientos sobre separación de responsabilidades, principios SOLID, inversión de dependencias y organización de las capas. Cuando estos conceptos no se comprenden, el proyecto puede presentar únicamente una división formal de carpetas, sin alcanzar la independencia esperada. En este escenario, la Clean Architecture aumenta la complejidad del código sin mejorar efectivamente su calidad.

De este modo, la pregunta de investigación se respondió al constatar que la Clean Architecture contribuye a la calidad y la evolución del software cuando se aplica en sistemas que poseen reglas de negocio complejas, múltiples integraciones, necesidad frecuente de cambios y expectativa de larga duración. Por otra parte, su adopción puede generar costes, dificultades y abstracciones innecesarias cuando se produce de manera rígida, indiscriminada o incompatible con las necesidades reales del proyecto.

También se comprobó que la adopción de la Clean Architecture no tiene que producirse de manera completa. Sus principios pueden aplicarse de forma parcial y progresiva, de acuerdo con el crecimiento y la complejidad del sistema. La separación entre dominio e infraestructura, por ejemplo, puede implementarse inicialmente, mientras que nuevas capas y abstracciones se añaden únicamente cuando surgen necesidades concretas de mantenimiento, pruebas o sustitución tecnológica.

Se concluye, por lo tanto, que la Clean Architecture no debe considerarse exclusivamente una aliada o una villana del desarrollo de software moderno. Representa una herramienta arquitectónica cuya efectividad depende del contexto, la planificación y la forma en que se utilizan sus principios. Cuando se aplica de manera crítica, proporcional y pragmática, puede contribuir a la construcción de sistemas más organizados, comprobables y preparados para evolucionar. Sin embargo, cuando se adopta únicamente como tendencia o modelo obligatorio, puede comprometer la simplicidad, elevar los costes y dificultar el proceso de desarrollo.

4 Referencias

BATISTA, M. V.; PINTO, G. S. Arquitetura de microfrontend. Revista Interface Tecnológica, v. 20, n. 2, p. 208–218, 2023. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1803.

CIRIANO, G.; PINTO, G. S. A evolução da arquitetura de software: do monólito aos microsserviços. Revista Interface Tecnológica, v. 22, n. 2, p. 122–133, 2025. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/2302.

DONATO, D. P.; ZUCHI, J. D. Arquitetura limpa: como minimizar o custo de um software. Revista Interface Tecnológica, v. 18, n. 2, p. 229–239, 2021. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1288.

ESCARELI, G. P.; MALARA, R. D. Manutenibilidade de software: ferramentas que auxiliam e garantem a qualidade. RECIMA21 – Revista Científica Multidisciplinar, v. 3, n. 12, p. e3122450, 2022. Disponível em: https://recima21.com.br/index.php/recima21/article/view/2450.

FERREIRA, V. B. S.; FERREIRA, C. A.; GRANDE, E. T. G. Estado da arte da pesquisa em: Clean Architecture e princípios de SOLID. Research, Society and Development, v. 11, n. 16, p. e335111637198, 2022. Disponível em: https://rsdjournal.org/rsd/article/download/37198/31702/418592.

GOMES, Y. L.; GIBERTONI, D. Web services: componentes reutilizáveis no processo de desenvolvimento orientado ao reuso. Revista Interface Tecnológica, v. 19, n. 2, p. 282–293, 2022. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1531.

GONÇALVES, B. P.; MENDES, O. L. Princípio da inversão de dependência na qualidade de software: aplicação da injeção de dependência no desenvolvimento de software. Revista Interface Tecnológica, v. 19, n. 1, p. 34–46, 2022. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1362.

KANIZAWA, D. T.; PINTO, G. S. Arquitetura de microsserviços. Revista Interface Tecnológica, v. 19, n. 2, p. 308–318, 2022. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1538.

LIMA, D. P. C.; LEAL, E. D.; MESQUITA, L. J. S. Benefícios da utilização da pirâmide de teste em uma aplicação mobile. Revista Interface Tecnológica, v. 20, n. 1, p. 63–75, 2023. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1607.

NASCIMENTO, C. H. P.; SOTTO, E. C. S. Microfrontend: um estudo sobre o conceito e aplicação no frontend. Revista Interface Tecnológica, v. 17, n. 1, p. 153–165, 2020. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/798.

PEGRUCCI, M.; TIOSSO, F.; REIS, H. M. Análise de um estudo de caso para comparar as arquiteturas monolítica e de microsserviços. Revista Interface Tecnológica, v. 17, n. 2, p. 325 –337, 2020. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/1029.

ROMANINI, I. Z.; SOTTO, E. C. S. Selenium Web Driver na evolução dos testes manuais. Revista Interface Tecnológica, v. 16, n. 2, p. 112–123, 2019. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/view/627.

SERAFIM, G. P.; DUARTE, A. M. D.; DUARTE, D. Análise quantitativa da aquisição de dívida técnica em sistemas legados. Revista de Sistemas e Computação, v. 12, n. 1, p. 10–19, 2022. Disponível em: https://revistas.unifacs.br/index.php/rsc/article/viewFile/7569/4507.

SILVA, R. A.; COSTA, J. M. Arquitetura modular e escalável para desenvolvimento de aplicativos móveis no Escritório de Soluções Criativas. JNT – Facit Business and Technology

Journal, v. 1, n. 55, p. 780–797, 2024. Disponível em: https://revistas.faculdadefacit.edu.br/index.php/JNT/article/download/3051/2082.

SILVA, R. C.; SAITO, R. Aplicando arquitetura de microsserviços no desenvolvimento de software. Revista de Ubiquidade, v. 4, n. 2, p. 58–84, 2021. Disponível em: https://revistas.anchieta.br/index.php/RevistaUbiquidade/article/view/1859.