

Patrones de Diseño

María Alejandra Marín Henríquez
Servicio Nacional de Aprendizaje – SENA

Resumen—RESUMEN

Este artículo se centra en un análisis detallado de veinte artículos científicos que nos hablan sobre los patrones de diseño, las arquitecturas más contemporáneas y los pilares del desarrollo de software. Y es que, de hecho, todo este estudio no fue en vano: se convirtió en la base teórica directa para el diseño de nuestra Plataforma de Gestión de Experiencias Significativas. Se analizan los patrones de comportamiento, estructurales y creacionales, además de arquitecturas como son Domain-Driven Design (DDD), N-Capas, Onion Architecture y MVC. Esto es crucial, ya que el proyecto se fundamenta en la arquitectura N-Capas para su backend (implementado en C#/.NET Core 8 con Entity Framework Core, SignalR y JWT Bearer) y evoluciona hacia DDD y Microservicios para garantizar la escalabilidad. Asimismo, tratan la seguridad del software, la automatización de pruebas, los microservicios, las tendencias futuras y los principios SOLID. Los patrones de diseño siguen siendo instrumentos primordiales para asegurar la escalabilidad, la calidad y el mantenimiento en sistemas complejos. También queda claro que su uso apropiado ayuda a crear un software seguro y reutilizable. En esencia, el proyecto es nuestro caso de prueba para validar que esta teoría funciona en la realidad del sector educativo.

Palabras Claves: Patrones de Diseño, pruebas de software, automatización, microservicios, MVC, SOLID y arquitectura de software.

ABSTRACT

This article centers on a detailed analysis of twenty scientific papers that discuss design patterns, contemporary architectures, and the pillars of software development. And, as it turns out, this entire study wasn't done in vain: it became the direct theoretical foundation for the design of our Significant Experiences Management Platform. The paper analyzes creational, structural, and behavioral patterns, in addition to architectures such as Domain-Driven Design (DDD), N-Tier, Onion Architecture, and MVC. This is crucial, as the project's backend is fundamentally built upon the N-Tier architecture (implemented in C#/.NET Core 8 with Entity Framework Core, SignalR, and JWT Bearer) and evolves toward DDD and Microservices to ensure scalability. Furthermore, the review covers software security, test automation, microservices, future trends, and the SOLID principles. Design patterns remain essential instruments for ensuring scalability, quality, and maintenance in complex systems. It's also clear that their appropriate use helps create software that is secure and reusable. In essence, the project serves as our practical case study to validate that this theory functions within the realities of the education sector.

Keywords: Design Patterns, software testing, automation, microservices, MVC, SOLID, and software architecture.

Index Terms—Patrones de Diseño, pruebas de software, automatización, microservicios, MVC, SOLID, arquitectura de software

I. INTRODUCCIÓN

El desarrollo de software moderno enfrenta desafíos cada vez más complejos. No basta con que un sistema funcione; debe ser mantenable, escalable y capaz de evolucionar sin colapsar bajo su propia complejidad. La ingeniería de software ha desarrollado, a lo largo de décadas, un conjunto de prácticas fundamentales para enfrentar estos retos: los patrones de diseño, las arquitecturas escalables y los principios de programación que buscan minimizar el acoplamiento y maximizar la cohesión. Lejos de ser simples abstracciones académicas, estos conceptos funcionan como herramientas prácticas que nos ayudan a organizar, entender y mejorar los sistemas que construimos día a día.

Colombia vive una transformación digital en su sector educativo que ha creado una necesidad real: plataformas robustas capaces de manejar información compleja sin quebrarse. De ahí nació la Plataforma de Gestión de Experiencias Significativas, un sistema pensado para documentar, evaluar y compartir prácticas pedagógicas innovadoras entre instituciones educativas. Pero este proyecto es más que una herramienta funcional; se convirtió en un laboratorio donde pudimos poner a prueba, en condiciones reales, todo lo que la teoría nos dice sobre patrones y arquitecturas.

Decidimos estructurar el sistema con una arquitectura N-Capas en C#/.NET Core, dividiendo el código en cuatro capas bien definidas: API para la presentación, Service para la lógica de negocio, Repository para el acceso a datos, y Entity para el modelo de dominio. Esta división no fue caprichosa. Responde a principios que, según nuestra experiencia, realmente facilitan el mantenimiento y las pruebas, además de preparar el terreno para evolucionar hacia arquitecturas más sofisticadas como Domain-Driven Design o microservicios cuando el proyecto lo requiera. Hoy el sistema tiene más de 38 servicios, 37 repositorios y está organizado en 5 módulos funcionales (Seguridad, Operación, Parámetros, Geográfico y Base), cada uno con su propia responsabilidad claramente marcada.

Los patrones de diseño que Gamma, Helm, Johnson & Vlissides documentaron en 1994 siguen siendo relevantes, aunque han pasado tres décadas. Investigadores más recientes como Sánchez (2023) o Blas, Leone & Gonnet (2019) han explorado cómo estos patrones se adaptan a temas modernos: calidad, automatización, seguridad.

Escalante (2013) y Bermúdez junto con sus colegas (2012) analizaron su aplicación en contextos empresariales y arquitectónicos. Nosotros nos apoyamos en todo ese conocimiento acumulado, pero con un objetivo concreto: mostrar que los patrones de diseño, aplicados con sentido común y no por dogma, pueden convertir un proyecto educativo en un sistema que realmente funciona a nivel profesional y que puede crecer sin desmoronarse.

II. MARCO TEÓRICO

Los patrones de diseño son soluciones reutilizables a problemas que surgen de manera constante en el desarrollo de software, con el propósito de proporcionar una base para poder realizar una actividad, mejorando la calidad del producto que esa actividad dé como resultado. En esencia, actúan como recetas de alta cocina para la codificación, evitando que el equipo reinvente soluciones ya probadas.

Hay patrones que abarcan las distintas etapas del desarrollo; desde el análisis hasta el diseño y desde la arquitectura hasta la implementación. En el caso de los patrones computacionales, un software estructurado y modulado posee una mejor calidad y es más sencillo corregir errores, implementar mejoras y actualizaciones. En sí, un patrón de diseño puede verse como una plantilla que puede ser aplicada en muchas situaciones diferentes”, para dar una buena solución.

Gamma, Helm, Johnson & Vlissides (1994) clasifican los patrones de diseño en tres categorías primordiales, y la Plataforma de Experiencias Significativas los incluye activamente para garantizar su robustez:

Patrones creacionales: Fábricas para la Persistencia. Singleton, Factory, Builder y Prototype originan diversos mecanismos de creación de los objetos, aumentando la flexibilidad y la reutilización del código. La Plataforma implementa la filosofía de Fábrica Abstracta a través del Patrón Repository: la capa Business solo interactúa con la interfaz del Repository para crear o recuperar objetos de experiencia, mientras que la implementación concreta (Entity Framework Core) se encarga de la base de datos (SQL Server). Esto desacopla la creación de datos de la lógica central y facilita cambiar de motor sin tocar la capa de negocio.

En el proyecto, el `ExperienceRepository` implementa la interfaz `IExperienceRepository`, heredando de `BaseModelRepository`. Lo interesante de esta jerarquía es que la capa de servicio solo conoce la abstracción, nunca la implementación concreta. Tomemos como ejemplo el método `GetByIdWithDetailsAsync`: por dentro hace consultas bastante complejas con varios `Include` de Entity Framework para traer datos relacionados, pero todo ese detalle de persistencia queda completamente oculto para las capas que lo usan.

La Plataforma también usa el Patrón Builder para armar objetos complejos de forma gradual. La clase `ExperienceBuilder` nos permite construir una entidad

`Experience` con todas sus relaciones (institución, documentos, objetivos, líneas temáticas, grados, poblaciones, historial) de una manera que se lee casi como lenguaje natural. Cada método (`WithBasicInfo`, `WithInstitution`, `WithDocuments`, y así sucesivamente) devuelve la misma instancia del builder, lo que permite ir encadenando llamadas una tras otra. Esto nos salvó de tener constructores con veinte o treinta parámetros, que serían imposibles de mantener. Al final, el método `Build()` nos entrega la entidad ya armada y lista para guardar en la base de datos.

Patrón Builder - Construcción de Objetos Complejos

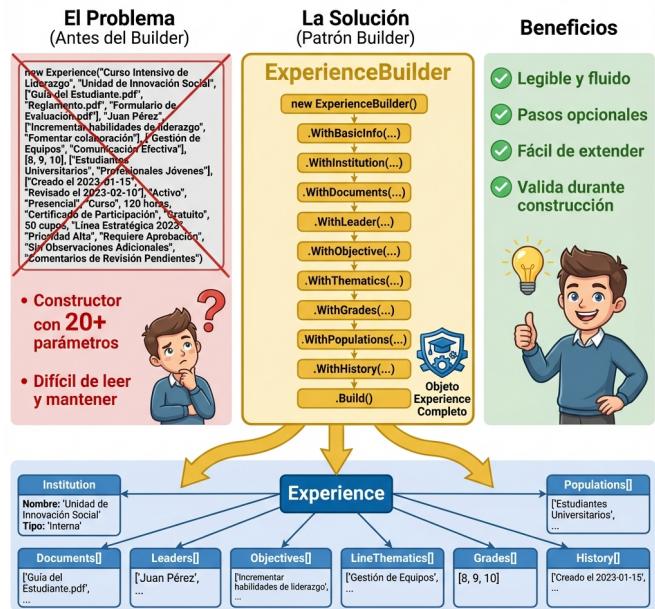


Figura 1: Patrón Builder aplicado a la construcción de entidades Experience complejas

Patrones estructurales: Simplificación y Seguridad. Patrones como Facade y Proxy nos ayudan a componer sistemas de manera eficiente. El Panel de Administración funciona como un Patrón Facade: presenta una interfaz sencilla que esconde toda la complejidad del subsistema de seguridad y autorización que hay detrás. Por otro lado, la autenticación con JWT Bearer actúa como Patrón Proxy: cada token decide si un usuario puede o no acceder a un recurso específico.

En código, el `AuthController` expone solo tres endpoints simples (`Login`, `UpdatePassword`, `RenewToken`), pero detrás de esa simplicidad hay validación de credenciales, generación de tokens JWT, hashing de contraseñas y renovación de sesiones. El controlador delega todo ese trabajo en `IAuthService`, que coordina con `IJwtAuthenticationService` y los repositorios de usuarios. Para el cliente que consume la API, todo se ve limpio y directo.

Patrones de comportamiento: Comunicación desacoplada. Se ocupan de la comunicación entre objetos. La Plataforma usa el Patrón Observer para notificaciones

internas: cuando el estado de una experiencia cambia (Sujeto), notifica a los Observadores (usuarios o roles interesados). La implementación práctica usa utilidades como MailKit para el envío de correos, desacoplando la lógica de negocio de la de envío.

Especificamente, el sistema implementa Observer mediante SignalR, una biblioteca de Microsoft para comunicación en tiempo real. Cuando se registra una nueva experiencia, el método `RegisterExperienceAsync` en `ExperienceService` invoca:

```
await _hubContext.Clients.All.SendAsync("ReceiveNotification",
{
    Title = "Nueva experiencia registrada",
    ExperienceName = experience.NameExperiences,
    CreatedBy = experience.User?.Person?.FirstName,
    Date = DateTime.Now
});
```

Este código notifica a todos los clientes conectados sin que el servicio conozca quiénes son ni cómo están implementados. De manera similar, el método `NotifyAdmins` envía notificaciones solo al grupo de administradores cuando una experiencia se actualiza. Esta implementación del patrón Observer permite agregar nuevos observadores (clientes web, móviles, sistemas externos) sin modificar la lógica de negocio.

Arquitecturas de alto nivel. Las arquitecturas de software proporcionan una estructura de nivel alto. En el método MVC (Modelo-Vista-Controlador), se puede distinguir con claridad entre la lógica de negocio, la presentación y el flujo de control. Este principio de segregación fundamenta la elección de la Plataforma:

- El backend se basa en la Arquitectura N-Capas (C#/.NET Core), que es una evolución directa de las arquitecturas de alto nivel. Esta elección establece la estructura modular que evita el acoplamiento y asegura que el sistema pueda crecer eficientemente (RNF-04).

La arquitectura en N-Capas organiza el sistema en capas lógicas: presentación, negocio, datos, etc. La Onion Architecture y el Domain-Driven Design (DDD) ponen el dominio del negocio en el centro, separándolo de los detalles tecnológicos externos. Además, los principios SOLID (Responsabilidad única, Segregación de interfaces, Sustitución de Liskov, Inversión de dependencias y Abierto/Cerrado) promueven un código más limpio, modular y siempre en cambio, por lo que complementan estos patrones arquitectónicos.

III. DESARROLLO

III-A. Patrones creacionales

Los patrones creacionales se enfocan en definir y gestionar la manera en que se crean los objetos. Su propósito es permitir la construcción de sistemas flexibles, desacoplados y capaces de operar independientemente de la manera en que los objetos son creados, compuestos o representados.

Un patrón de creación de clases utiliza la herencia para determinar la clase que debe estar instanciada como ejemplo se puede citar el *Factory Method*, y el patrón de creación de objetos delega la instanciación en otro objeto como ejemplo se cita el patrón *Builder*, como lo señala García-Peña.

El patrón *Factory* es una herramienta poderosa para crear objetos de manera flexible y centralizada. Su valor está en reducir el acoplamiento y facilitar la extensión de funcionalidades, aunque exige cuidado para no caer en una complejidad innecesaria. Bien aplicado, se convierte en un aliado clave para aplicaciones que buscan escalabilidad y mantenimiento a largo plazo.

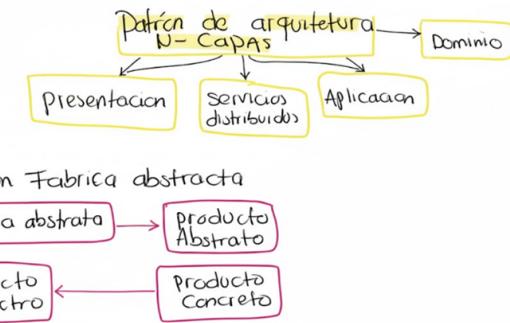


Figura 2: Aplicación del Patrón Factory para desacoplar la creación de objetos en la plataforma

III-A1. Patrón Builder en la Plataforma de Experiencias Significativas: El patrón *Builder* se vuelve especialmente útil cuando trabajamos con objetos que necesitan varios pasos para construirse o que tienen muchas propiedades opcionales. En nuestro caso, la entidad `Experience` representa una experiencia pedagógica significativa y está conectada con múltiples entidades: la institución educativa, documentos de soporte, objetivos específicos, líneas temáticas, grados escolares, poblaciones beneficiarias e historial de cambios. Si intentáramos construir manualmente una instancia completa de `Experience`, terminaríamos con un constructor de más de veinte parámetros. Sería un desastre ilegible y lleno de errores potenciales.

La clase `ExperienceBuilder` resuelve este problema aplicando el patrón *Builder*. El constructor inicializa una instancia de `Experience` con valores por defecto (`State = true, CreatedAt = DateTime.UtcNow`). Después, cada método del builder (`WithBasicInfo, WithInstitution, WithDocuments, WithObjective, etc.`) configura una parte específica del objeto y devuelve `this`, lo que permite encadenar llamadas de forma fluida. Al final, el método `Build()` nos entrega la entidad completamente armada.

Tomemos como ejemplo el método `WithObjective`: recibe una colección de `ObjectiveCreateRequest` y, para cada uno, usa otro builder (`ObjectiveBuilder`) para construir los objetivos con sus soportes y monitoreos asociados. Esta composición de builders nos permite manejar estructuras jerárquicas complejas

de manera elegante. El código resultante en `ExperienceService.RegisterExperienceAsync` queda notablemente limpio:

```
var experience = new ExperienceBuilder()
    .WithBasicInfo(request)
    .WithInstitution(request.Institution)
    .WithDocuments(request.Documents)
    .WithDevelopment(request.Developments)
    .WithLeader(request.Leaders)
    .WithObjective(request.Objectives)
    .WithThematics(request.ThematicLineIds)
    .WithGrades(request.Grades)
    .WithPopulations(request.PopulationGradeIds)
    .WithHistory(request.HistoryExperiences, request)
    .Build();
```

Esta implementación del patrón Builder nos trajo varios beneficios concretos: primero, la legibilidad del código mejoró drásticamente porque cada método describe claramente qué aspecto del objeto está configurando; segundo, ganamos flexibilidad ya que podemos omitir pasos opcionales sin afectar la construcción; tercero, el mantenimiento se simplificó porque agregar nuevas propiedades solo requiere añadir un nuevo método `With...` sin tocar los existentes (principio Open/Closed de SOLID). Además, centralizar la lógica de construcción en el builder nos facilita aplicar validaciones o transformaciones de datos antes de asignarlos a la entidad.

III-B. Patrones estructurales

Se centran en la manera de cómo las clases y los objetos se unen para crear estructuras complejas. Un patrón estructural de clases utiliza la herencia para componer interfaces o implementaciones, por ejemplo, el patrón *Adapter*. Un patrón estructural de objetos describe la forma en que se componen objetos para obtener nueva funcionalidad, además se añade la flexibilidad de cambiar la composición en tiempo de ejecución, lo cual no es posible con la composición de clases estáticas; como ejemplo de este tipo de patrones se puede mencionar al patrón *Composite*.

III-B1. Generación Automática de Interfaces a Partir de Patrones Estructurales de Tareas: El uso a partir de patrones estructurales de tareas permite automatizar la generación de interfaces, mejorando así la eficiencia del desarrollo. Sin embargo, la dependencia de reglas predefinidas y la complejidad de los diagramas CTT los cuales pueden llegar a convertirse en ilegibles para aplicaciones medianas. Para esto, el proceso de generación de los CTT debe ser oculto para el analista. Se propuso como solución el uso de bocetos (*SKETCHES*) para generar los diagramas. Gracias a estas reglas los analistas están limitados a utilizar las reglas existentes. Esta limitación afecta a la construcción de los bocetos.

III-C. Patrones de comportamiento

Este tipo de patrones tiene que ver con la asignación de responsabilidades entre objetos y los algoritmos. Además

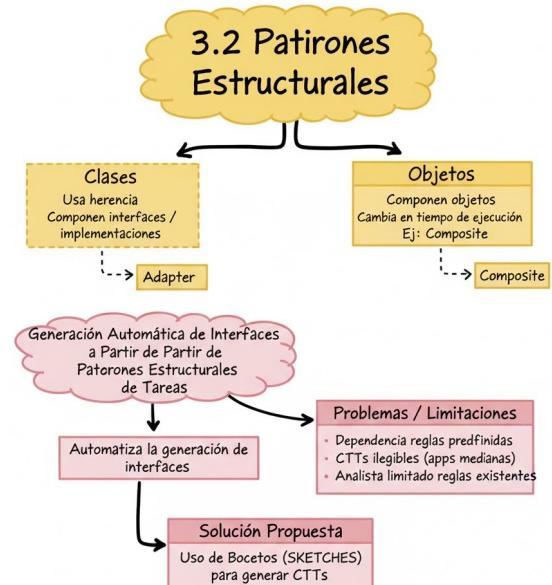


Figura 3: Ilustración del Patrón Composite para componer objetos flexiblemente

de clases y objetos, siendo patrones de comunicación entre ellos. Los patrones de comportamiento de clases se valen de la herencia para repartir el comportamiento entre las clases. Un ejemplo es el patrón *Template Method*.

Por otra parte, los modelos de conducta que se basan en objetos utilizan la composición en vez de la herencia para coordinar un grupo de objetos que cooperan para llevar a cabo tareas que uno solo no podría resolver por sí mismo. Un patrón típico de esta categoría es el patrón *Observer*.

III-D. Extracción automatizada de contenido web

La recolección de información de uno o más sitios web de manera automatizada, emulando la interacción entre un usuario y un servidor, se basa en el análisis de estructuras HTML y nos indica que no requiere de la autorización de propiedades.

El uso de las estructuras repetitivas o plantillas facilita la funcionalidad de un programa informático que extrae contenido web; dicha intrusión genera un incremento considerable en el uso de recursos, considerando la permanente ejecución de instrucciones para obtener tanto contenido como sea posible.

Para reducir la vulnerabilidad de los sitios web frente a procesos de extracción de contenido masivo, en el artículo se planteó un patrón de diseño de software tomando como referencia el patrón *Template View* de Martin Fowler, al cual se le agregó una capa de aleatorización que permita generar estructuras HTML no predecibles.

Esto se lleva a cabo mediante la aplicación de una herramienta de extracción de contenido a un sitio web de prueba, cuya capa de presentación se desarrolló tomando

en cuenta el patrón de diseño que fue propuesto; se logró verificar una reducción considerable de la calidad de datos extraídos.

III-E. Arquitecturas de software relacionadas con patrones

Es un diseño integral que permite formular diseños de arquitectura de software destinadas a la representación de aplicaciones web. Este entorno abstrae los principales problemas identificados a diseño de calidad. Para este se utiliza como base un metamodelo (un modelo que describe cómo se construyen otros modelos) para componentes arquitectónicos que identifican un conjunto de elementos comúnmente utilizados en ciertas arquitecturas. Sobre el modelo se construye una herramienta de instanciación gráfica que se complementa con la verificación de patrones de diseño a fin de garantizar su correcta aplicación.

Aplicación en la Plataforma. El backend en C#/.NET Core se apoya en la arquitectura N-Capas como base modular, y evoluciona hacia DDD y microservicios para absorber incrementos de carga. El Patrón Repository aísla la lógica de negocio de la persistencia (SQL Server), permitiendo reemplazar el motor de datos sin afectar el dominio y preparando la transición a servicios independientes.

Los patrones arquitectónicos estructuran los sistemas en niveles macro. Entre los principales se destacan:

III-E1. Paquete Java para la integración de herramientas de pruebas de software basado en patrones de software: Los patrones de diseño de arquitectura son soluciones para problemas recurrentes en el desarrollo; al aplicar un patrón incorrectamente este puede llegar a generar problemas de rendimiento, seguridad o mantenibilidad.

Verificar que el patrón esté correctamente implementado es fundamental para garantizar la calidad del software. Algunas metodologías propuestas son:

- Modelado formal del patrón: uso de diagramas y especificaciones formales para describir la estructura y el comportamiento esperado; un ejemplo es el patrón *Layered Architecture* modelado en UML.
- Transformación del modelo en un formato verificable: conversión de diagramas a un lenguaje formal como Alloy para su verificación automática.
- Verificación de propiedades: definir si el patrón cumple con ciertas propiedades como responsabilidad única, inversión de dependencias, etc.

Esta metodología permite detectar errores en implementaciones que a simple vista parecen correctas. La formalización ayuda a tener documentación clara y no ambigua del patrón, y la verificación automática ayuda a la reducción de riesgos y asegura la calidad desde las primeras etapas.

III-E2. Arquitectura Modelo-Vista-Controlador (MVC): La arquitectura Modelo-Vista-Controlador (MVC) es una de las que más se destacan. Este patrón busca abordar ventajas y desventajas, la facilidad

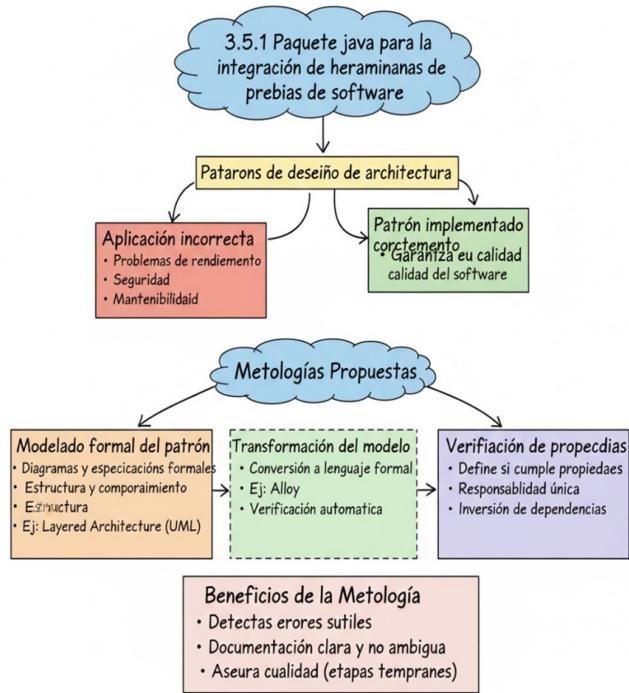


Figura 4: Flujo de verificación automática de patrones para reducir riesgos en implementaciones

en el desarrollo, la alta mantenibilidad y el alto aprovechamiento de código para que las empresas puedan alcanzar estos objetivos. En el proceso de modelo el patrón MVC requiere más tiempo para analizar y modelar el sistema, lo que se recomienda para proyectos pequeños, pero es muy esencial para proyectos grandes cuyo control y organización del desarrollo se obtienen de manera eficiente y satisfactoria.

Uno de los patrones básicos para el desarrollo de aplicaciones orientadas a la web es el modelo vista controlador (MVC), el cual permite hacer una separación en tres partes gráficas de la aplicación y los procesos de esta misma.

La evolución del desarrollo web hace énfasis en la influencia de tecnologías como AJAX, FLEX y Openlaszlo. Finalmente, está la posibilidad de la unificación de todas las tecnologías para llegar a la creación de aplicaciones web que funcionen completamente a favor del cliente, y que todos estos procesos complejos actuales que tiene la arquitectura Cliente-Servidor se reduzcan a la simple toma de información que se encuentran en servidores de bases de datos.

Implementación de un framework para el desarrollo de aplicaciones web. Aborda el uso de las metodologías ágiles en el desarrollo de software, principalmente enfatizado en Scrum como un marco de trabajo y así mejorar la gestión y la calidad de los proyectos. Scrum se determina como un método basado en iteraciones cortas (sprints), siendo unos roles bien definidos (Product

Owner, Scrum Master y Equipo de Desarrollo) y eventos importantes como la planificación, las reuniones diarias, la revisión y la retrospectiva. También el uso del Product Backlog y Sprint Backlog para organizar y priorizar tareas.

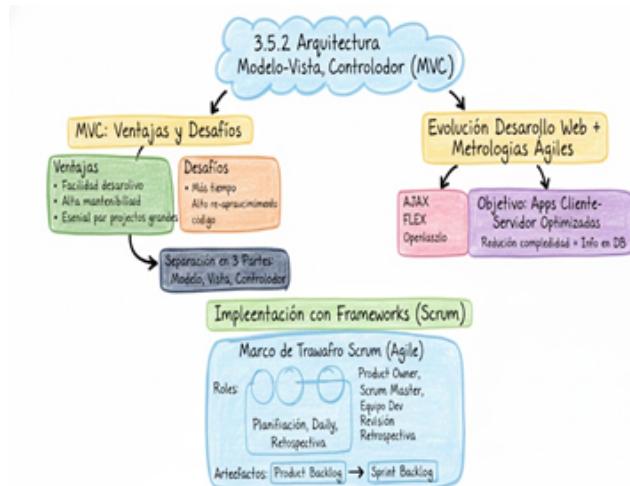


Figura 5: Marco Scrum: roles, iteraciones cortas y artefactos (Product/Sprint Backlog)

III-E3. Arquitectura N-Capas: La arquitectura en N-Capas está distribuida en capas como son las de presentación, aplicación, dominio y datos. Esta estructura permite tener el control sobre responsabilidades, facilitando el mantenimiento y la modularidad.

Siendo el patrón N-Capas orientado en el dominio y la fábrica abstracta donde se diseñan aplicaciones empresariales. Estos patrones ayudan a que el software sea más organizado, flexible y fácil de mantener.

El backend se basa en la Arquitectura N-Capas (C#/ .NET Core), que es una evolución directa de las arquitecturas de alto nivel. Esta elección establece la estructura modular que evita el acoplamiento y asegura que el sistema pueda crecer eficientemente (RNF-04).

Implementación concreta de N-Capas en la Plataforma. En la práctica, esta arquitectura se traduce en cuatro proyectos .NET bien separados: API, Service, Repository y Entity. Cada capa sabe exactamente qué le toca hacer y solo habla con las capas que tiene al lado, respetando esa idea de que las dependencias van en una sola dirección.

La capa API tiene los controladores que exponen los endpoints HTTP. Tomemos `AuthController` como ejemplo: maneja todo lo relacionado con autenticación. Pero esta capa es bien delgada; solo recibe requests, valida que el formato esté correcto, llama a los servicios correspondientes y devuelve las responses HTTP. Nada de lógica de negocio aquí.

La capa Service es donde vive toda la lógica de negocio. `ExperienceService`, por ejemplo, orquesta todo el proceso de registrar una experiencia: valida que se cumplan las reglas del negocio, coordina varios repositorios, trans-



Figura 6: Arquitectura N-Capas adoptada en el backend de la Plataforma de Gestión de Experiencias Significativas

forma datos cuando hace falta y dispara通知aciones. Lo importante es que esta capa trabaja con interfaces de repositorios (como `IExperienceRepository`), nunca con las implementaciones concretas. Así aplicamos Inversión de Dependencias.

La capa `Repository` se encarga del acceso a datos. `ExperienceRepository` implementa `IExperienceRepository` y esconde todas las consultas a la base de datos detrás de Entity Framework Core. Métodos como `GetByIdWithDetailsAsync` arman consultas complejas con varios `Include`, pero esa complejidad nunca se filtra hacia las capas de arriba.

Finalmente, la capa `Entity` define el modelo de dominio: las entidades, los DTOs, los requests y el contexto de base de datos. Lo bueno es que esta capa no depende de ninguna otra, así que es completamente reutilizable.

Un flujo completo ilustra la interacción entre capas. Cuando un usuario registra una experiencia:

1. `ExperienceController` (API) recibe el POST request con `ExperienceCreateRequest`
2. El controlador invoca `_experienceService.RegisterExperienceAsync(request)`
3. `ExperienceService` (Service) usa `ExperienceBuilder` para construir la entidad
4. El servicio llama `_experienceRepository.AddAsync(experience)`
5. `ExperienceRepository` (Repository) persiste en la base de datos vía EF Core
6. El servicio dispara notificación SignalR a través de `_hubContext`
7. El controlador retorna `ApiResponseRequest` con el resultado

Esta separación nos dio beneficios reales y medibles. Durante el desarrollo cambiaron de SQL Server a PostgreSQL, y no tuvimos que tocar ni una línea de código en Service o API; solo ajustamos la configuración del contexto

en Entity. Las pruebas unitarias de los servicios corren con repositorios mock, sin necesidad de levantar una base de datos. Y cuando llegue el momento de migrar a microservicios, cada módulo (Seguridad, Operación, Parámetros) se puede extraer como un servicio independiente sin perder su estructura interna de capas.

Arquitectura N-Capas - Plataforma de Experiencias Significativas

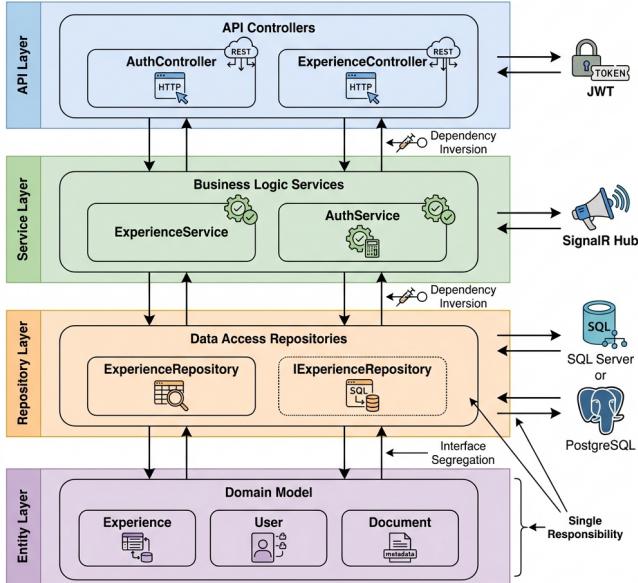


Figura 7: Arquitectura N-Capas implementada con las 4 capas y sus dependencias

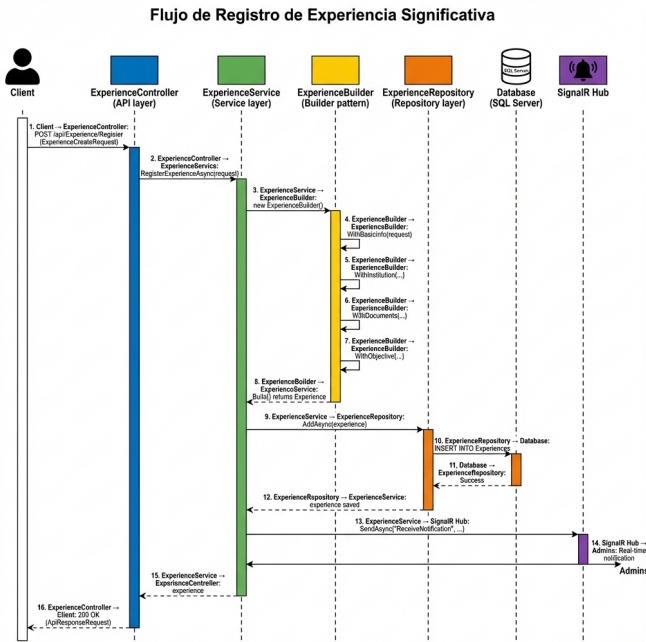


Figura 8: Diagrama de secuencia del flujo completo de registro de experiencia

Patrón Capas: el patrón arquitectónico Capas ayuda con la estructuración de las aplicaciones que se pueden descomponer en subtareas, en la que cada grupo está en un nivel particular de abstracción.

Patrón Fábrica Abstracta: es una clase que proporciona una interfaz para producir una familia de objetos.

Las clases que participan en este patrón son:

- **Fábrica Abstracta:** una interfaz para operaciones que crean objetos productos abstractos.
- **Fábrica Concreta:** implementa las operaciones para crear objetos productos concretos.
- **Producto Abstracto:** declara una interfaz para un tipo de objeto producto.
- **Producto Concreto:** define un objeto producto para que sea creado por la fábrica correspondiente implementando la interfaz producto abstracto.

III-E4. Domain-Driven Design: Domain-Driven Design (DDD) diseña y desarrolla aplicaciones empresariales siguiendo principios como crear un lenguaje ubicuo para evitar ambigüedades, modelar entidades, agregados, repositorios, servicios de dominio, eventos de dominio y *value objects*. También evalúa patrones estratégicos como *Bounded Context* y *Context Mapping* para organizar mejor las responsabilidades y dependencias del sistema.

Cuando usamos DDD en microservicios, vemos que si el software refleja fielmente el negocio, los proyectos ganan claridad, comunicación y capacidad de crecer. Sí, exige más aprendizaje y esfuerzo al principio, pero los resultados en claridad y escalabilidad hacen que valga la pena.

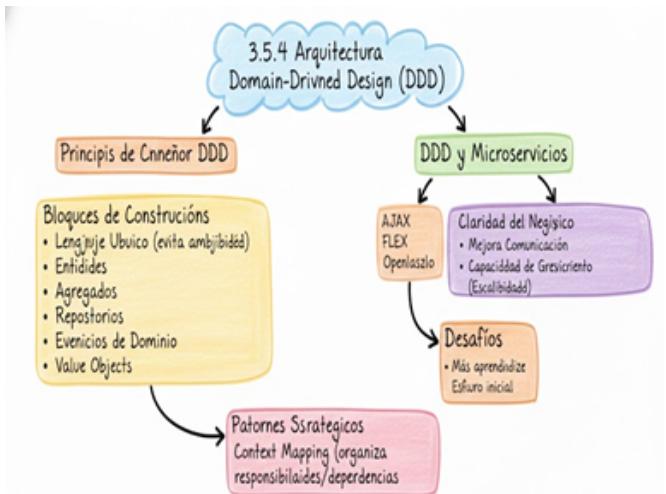


Figura 9: Evolución hacia microservicios basada en DDD para escalar con claridad de dominio

III-E5. Onion Architecture: La Onion Architecture destaca por su integración con el enfoque DDD para lograr sistemas más escalables, fáciles de mantener y con poco acoplamiento. Esta arquitectura organiza el código en capas concéntricas donde las dependencias siempre

apuntan hacia el centro, que es donde viven el dominio y las reglas de negocio.

Integrar DDD con Onion Architecture refuerza la separación de responsabilidades y la claridad en el modelo de dominio. En nuestro caso, diseñamos la aplicación dividiendo las funciones en capas y respetando la dirección de dependencias, lo que facilitó las pruebas unitarias y la incorporación de cambios.

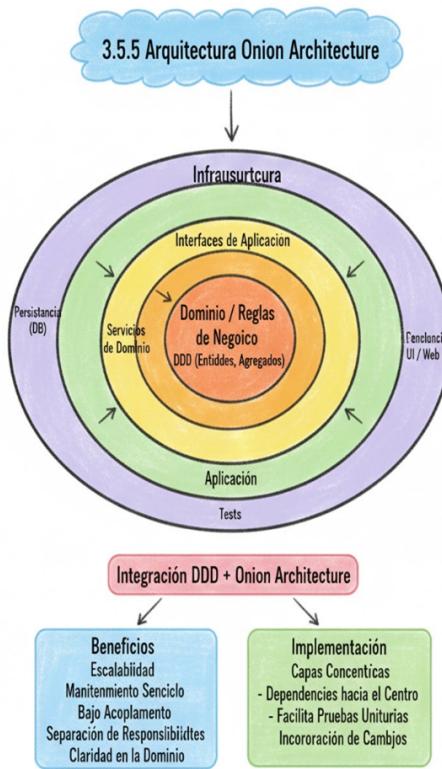


Figura 10: Organización concéntrica de la Onion Architecture y dirección de dependencias

III-F. Principios SOLID

Hay proyectos que proponen generadores de código para entornos empresariales, y muchos recurren a estrategias de calidad como los principios SOLID (*Single responsibility, Open-closed, Liskov substitution, Interface segregation y Dependency inversion*). El objetivo de estos principios es contribuir a la calidad del software y promover cierta estandarización en la arquitectura, influyendo tanto en el diseño como en la implementación. Al final, se trata de construir software de calidad sin depender completamente de que un programador específico esté siempre disponible.

Aplicación en la Plataforma de Experiencias Significativas. El sistema no solo cumple con SOLID; estos principios están integrados en su estructura base (N-Capas):

- **Single Responsibility:** Los roles y las clases de negocio se mantienen pequeñas y enfocadas; evitamos el temido

"Dios Object". Por ejemplo: el Profesor solo registra experiencias; el Evaluador solo califica.

- **Interface Segregation:** El menú dinámico por rol (HU33) le muestra a cada perfil solo las opciones que le corresponden (Administrador, Profesor, Evaluador), evitando que dependan de métodos que no necesitan.
- **Dependency Inversion:** La capa de negocio depende de la abstracción del repositorio (Repository), no de la implementación concreta (EF Core). Esto nos permite cambiar el motor de base de datos sin tocar la lógica central.
- **Open/Closed y Liskov:** Adoptar N-Capas y DDD nos permite extender con nuevos tipos de experiencias o reportes sin modificar los núcleos existentes, y las interfaces garantizan que podamos sustituir implementaciones de forma segura.

Ejemplos concretos de SOLID en el código. Cada principio se materializa en decisiones arquitectónicas específicas:

Single Responsibility Principle (SRP): Cada clase tiene una única razón para cambiar. `ExperienceService` se encarga exclusivamente de la lógica de negocio de experiencias (registro, actualización, consulta), mientras que `ExperienceRepository` maneja únicamente la persistencia. La responsabilidad de notificaciones se delega a `SignalR (IHubContext)`, y la generación de PDFs a `ExperiencePdfGenerator`. Esta separación facilita el testing: se pueden probar las reglas de negocio sin base de datos, y las consultas sin lógica de negocio.

Open/Closed Principle (OCP): El sistema está abierto a extensión pero cerrado a modificación. La clase base `BaseModelRepository< TEntity, TDto, TRequest >` define operaciones CRUD genéricas. Cuando necesitamos un repositorio para una nueva entidad, simplemente heredamos de esta clase base sin tocarla. Por ejemplo, `ExperienceRepository` extiende `BaseModelRepository` y agrega métodos específicos como `GetByIdWithDetailsAsync`, pero no modifica el comportamiento base.

Liskov Substitution Principle (LSP): Cualquier instancia de una clase derivada debe poder sustituir a su clase base sin alterar el comportamiento del programa. En la Plataforma, todos los repositorios implementan sus interfaces correspondientes (`IExperienceRepository`, `IUserRepository`, etc.) y heredan de `BaseModelRepository`. Los servicios trabajan con las interfaces, nunca con las implementaciones concretas, lo que nos permite sustituir repositorios por mocks en pruebas o por implementaciones alternativas sin romper nada.

Interface Segregation Principle (ISP): Los clientes no deben depender de interfaces que no usan. En lugar de tener una interfaz monolítica `IRepository` con todos los métodos posibles, el proyecto define interfaces específicas por entidad: `IExperienceRepository` expone solo los métodos relevantes para experiencias, `IAuthRepository` solo

los de autenticación. Así evitamos que AuthService tenga acceso a métodos de experiencias que no debería usar.

Dependency Inversion Principle (DIP): Los módulos de alto nivel no dependen de módulos de bajo nivel; ambos dependen de abstracciones. En `Program.cs`, la inyección de dependencias registra interfaces y sus implementaciones:

```
builder.Services.AddScoped<IExperienceRepository,
    ExperienceRepository>();
builder.Services.AddScoped<IExperienceService,
    ExperienceService>();
```

`ExperienceService` declara dependencias en su constructor como `IExperienceRepository`, no como `ExperienceRepository` concreto. El contenedor de inyección de dependencias resuelve estas abstracciones en tiempo de ejecución. Esto nos permite cambiar la implementación del repositorio (por ejemplo, de Entity Framework a Dapper) sin tocar ni una línea de código en el servicio.

Patrón Repository - Arquitectura de Acceso a Datos

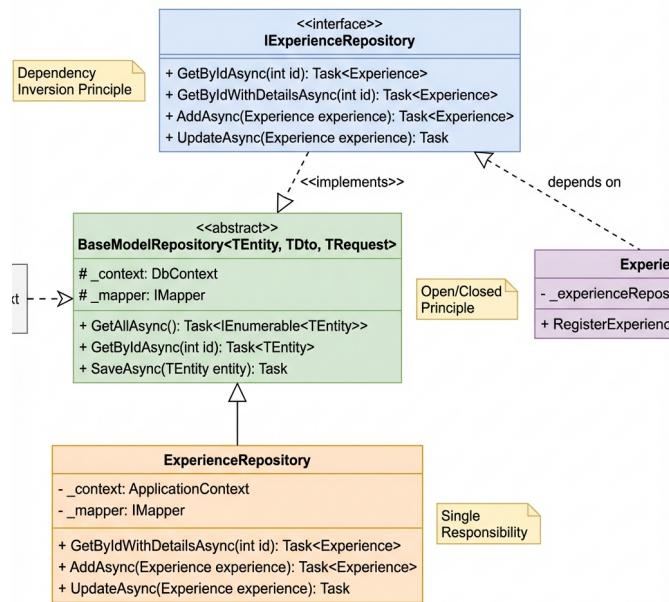


Figura 11: Diagrama de clases del patrón Repository con jerarquía de interfaces

Los principios SOLID:

S: Single Responsibility: Principio de responsabilidad única.

O: Open/Closed: Principio Abierto – Cerrado.

L: Liskov Substitution: Principio de sustitución.

I: Interface Segregation: Principio de segregación de interfaz.

D: Dependency Inversión: Principio de inversión de dependencia.

III-F1. Principio SOLID “Automatización de Pruebas de software”: La automatización de pruebas de interfaces web utilizando Selenium WebDriver y Java, con el fin de

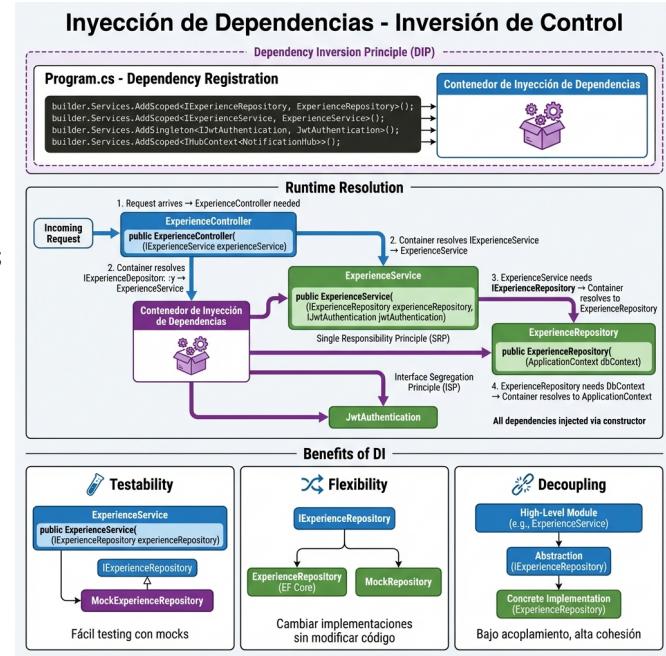


Figura 12: Inyección de dependencias y principio de inversión de control

conseguir un código más comprensible, mantenible y escalable. Siendo cada principio una Responsabilidad Única, Abierto/Cerrado, Sustitución de Liskov, Segregación de Interfaces e Inversión de Dependencias, se adaptan a un framework de pruebas, identificando violaciones comunes de estos y proponiendo refactorizaciones.

La aplicación del principio SOLID en automatización mejora la estructura, la claridad y la extensibilidad de las pruebas. Por ejemplo:

Single Responsibility: evita clases de prueba gigantes. Interface Segregation: permite usar solo métodos necesarios para cada prueba.

Dependency Inversión: permite usar *mocks, stubs y fakes* fácilmente.

Herramientas como el patrón *Page Object Model* y *Factory* aportan separación de responsabilidades y centralización de cambios de la interfaz. Estos abordan casos prácticos como la separación de objetos y acciones en clases distintas, adaptan el menú según su rol y dividen interfaces para evitar dependencias innecesarias; también aprovechan el uso de Java para la versatilidad de Selenium.

Los resultados muestran que aplicar SOLID mejora la calidad del código y facilita la adaptación a cambios sin comprometer la estabilidad, lo que garantiza la reducción de costo en mantenimiento del software, asegurando pruebas más eficientes y un desarrollo continuo de calidad.

III-G. Patrones de Diseño GOF (The Gang of Four)

En los procesos de desarrollo de software orientados a la web, inicialmente se construye un conjunto de criterios para evaluar y seleccionar procesos de desarrollo formales

de gran envergadura. Se establece el tamaño de la muestra para aplicar los criterios con estricto rigor metodológico; se inicia la inspección del código fuente para identificar el uso de patrones de diseño, llevando a cabo un proceso que permite la identificación de los patrones de diseño que son utilizados por expertos del área de la ingeniería del software.

The Gang of Four (GOF) dentro de procesos de desarrollo de software para la web definieron criterios para evaluar y seleccionar procesos de desarrollo grandes y formales. Se concluyó el tamaño de la muestra y se aplicaron los criterios de forma precisa; posteriormente, se revisó el código fuente para identificar si se utilizan patrones de diseño y se continuó con un proceso para identificar cuáles eran aplicados por expertos en ingeniería de software.

III-H. Patrones de diseño en la ingeniería de software

Estos patrones de diseño resuelven problemas comunes en el tema de desarrollo. Son soluciones ya probadas que facilitan crear software más organizado, fácil de mantener y ampliar. Los patrones se dividen en tres grupos principales: creacionales, estructurales y de comportamiento.

Adapter: permite que dos cosas que no son compatibles trabajen juntas.

Bridge: separa la abstracción de la implementación para que evolucionen de forma independiente.

Composite: trata igual a objetos individuales y a grupos de objetos.

Decorator: añade funciones extra a un objeto sin cambiar su estructura.

Façade: crea una interfaz simple para el uso de sistemas complejos.

Flyweight: libera espacio de memoria mediante la compartición de datos comunes entre objetos.

Proxy: opera como mediador para regular el acceso a un objeto.

Con estos patrones hay que manejarlos con bastante cuidado porque, aunque son muy útiles, utilizarlos sin necesidad complica innecesariamente el código.

III-I. Patrones de Diseño de Software

Estos patrones solucionan los problemas más comunes en el desarrollo de software, así evitando duplicar el código y permitiendo su reutilización. El estudio se centra en analizar la estructura, los componentes, las ventajas y desventajas de estos cinco patrones específicos: *Template Method*, *Model-View-Controller* (MVC), *Model-View-Presentar* (MVP), *Front Controller* y *Model-View-ViewModel* (MVVM).

IV. IMPLEMENTACIÓN

Implementamos la Plataforma de Gestión de Experiencias Significativas usando tecnologías modernas y probadas en entornos empresariales. La elección del stack tecnológico no fue al azar; cada componente lo seleccionamos

pensando en escalabilidad, soporte de la comunidad, compatibilidad con patrones de diseño y facilidad de mantenimiento a largo plazo.

IV-A. Stack Tecnológico

Backend: Desarrollamos el núcleo del sistema en C# con .NET Core 8, la versión LTS (Long Term Support) más reciente al momento del desarrollo. Esta plataforma ofrece rendimiento superior, soporte multiplataforma (Windows, Linux, macOS) y un ecosistema maduro de bibliotecas. La arquitectura N-Capas se materializa en cuatro proyectos .NET independientes pero interconectados: `API.csproj`, `Service.csproj`, `Repository.csproj` y `Entity.csproj`.

Persistencia: Entity Framework Core 8 actúa como ORM (Object-Relational Mapper), abstrayendo las operaciones de base de datos y permitiéndonos trabajar con objetos C# en lugar de SQL directo. El sistema soporta múltiples motores de base de datos: SQL Server como motor principal, PostgreSQL como alternativa open-source, y MySQL (actualmente deshabilitado pero listo para activarse cuando se necesite). Esta flexibilidad la logramos mediante contextos de base de datos separados (`ApplicationContext`, `ApplicationContextPostgres`) y cadenas de conexión configurables en `appsettings.json`. Las migraciones de base de datos se gestionan automáticamente al iniciar la aplicación mediante `context.Database.Migrate()`.

Autenticación y Autorización: La seguridad la implementamos con JWT (JSON Web Tokens) Bearer. Cuando un usuario se autentica exitosamente, el sistema genera un token firmado que contiene claims (roles, identificador de usuario, fecha de expiración). Este token debe incluirse en el header `Authorization` de cada request subsecuente. La clase `JwtAuthentication` encapsula la lógica de generación y validación de tokens, configurada como Singleton en el contenedor de inyección de dependencias. Los endpoints protegidos utilizan el atributo `[Authorize]`, mientras que los públicos (como login) usan `[AllowAnonymous]`.

Comunicación en Tiempo Real: SignalR, biblioteca de Microsoft para WebSockets, habilita notificaciones push del servidor a los clientes. El `NotificationHub` se mapea en `/hubs/notifications` y nos permite enviar mensajes a todos los clientes conectados (`Clients.All`) o a grupos específicos (`Clients.Group("admins")`). Esta implementación del patrón Observer desacopla completamente la lógica de negocio de la entrega de notificaciones.

Almacenamiento de Archivos: Supabase Storage gestiona documentos y PDFs generados. La clase `SubeBaseExperienceStorage` encapsula las operaciones de carga y descarga, exponiendo métodos como `UploadExperiencePdfToSupabase`. Esta abstracción nos permite cambiar el proveedor de almacenamiento (a AWS S3, Azure Blob Storage, etc.) sin modificar la lógica de negocio.

Generación de Documentos: QuestPDF, biblioteca para generación programática de PDFs, crea reportes de experiencias con formato profesional. `ExperiencePdfGenerator.Generate` recibe una entidad `Experience` completa y produce un documento PDF con todas las secciones: información básica, institución, objetivos, desarrollo, evaluaciones. El logo institucional se carga dinámicamente desde una URL configurada en `PdfSettings`.

Arquitectura Modular - 5 Módulos Funcionales

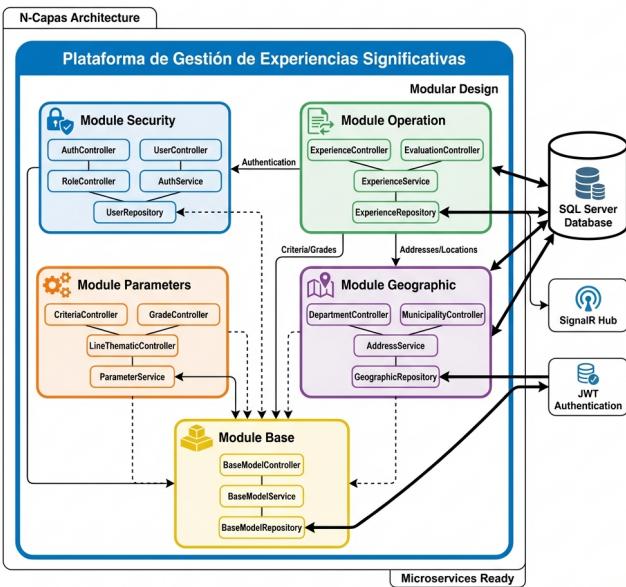


Figura 13: Arquitectura modular con 5 módulos funcionales independientes

IV-B. Flujo de Implementación: Registro de Experiencia

Para ilustrar cómo los patrones y la arquitectura trabajan en conjunto, analizamos el flujo completo de registro de una experiencia significativa:

1. Capa de Presentación (API): El cliente envía un POST request a `/api/Experience/Register` con un JSON que contiene `ExperienceCreateRequest`. El `ExperienceController` recibe el request, valida el modelo mediante Data Annotations, y delega al servicio:

```
var experience = await _experienceService
    .RegisterExperienceAsync(request);
return Ok(new ApiResponseRequest<Experience>(experience,
    true, "Experiencia registrada exitosamente"));
```

2. Capa de Negocio (Service):

`ExperienceService.RegisterExperienceAsync` orquesta el proceso completo. Primero, utiliza el patrón Builder para construir la entidad compleja:

```
var experience = new ExperienceBuilder()
    .WithBasicInfo(request)
```

```
.WithInstitution(request.Institution)
.WithDocuments(request.Documents)
.WithObjective(request.Objectives)
.Build();
```

Luego persiste la entidad invocando el repositorio, y finalmente dispara una notificación SignalR a todos los administradores conectados.

3. Capa de Datos (Repository): `ExperienceRepository.AddAsync` recibe la entidad construida y la agrega al contexto de Entity Framework:

```
_context.Experiences.Add(experience);
await _context.SaveChangesAsync();
```

Entity Framework traduce esta operación en sentencias SQL INSERT para la tabla `Experiences` y todas sus tablas relacionadas (documentos, objetivos, líneas temáticas, etc.), gestionando automáticamente las claves foráneas y transacciones.

4. Notificación (Observer): El servicio notifica a los observadores sin conocer su identidad:

```
await _hubContext.Clients.All.SendAsync("ReceiveNotification",
    new {
        Title = "Nueva experiencia registrada",
        ExperienceName = experience.NameExperiences
    });
});
```

Los clientes web conectados al hub reciben la notificación instantáneamente y actualizan su interfaz.

IV-C. Métricas del Proyecto

El proyecto alcanzó una escala considerable que valida la efectividad de los patrones aplicados:

- Controladores:** 38 controladores distribuidos en 5 módulos (Seguridad, Operación, Parámetros, Geográfico, Base)
- Servicios:** 39 servicios de negocio, cada uno con responsabilidad única
- Repositorios:** 37 repositorios de acceso a datos, todos heredando de `BaseModelRepository`
- Entidades:** Más de 40 entidades de dominio con sus respectivos DTOs y Requests
- Builders:** 4 builders para construcción de objetos complejos (Experience, Institution, Objective, Evaluation)
- Soporte Multi-BD:** 3 motores de base de datos soportados sin duplicación de código

La aplicación de patrones permitió que un equipo pequeño gestionara esta complejidad sin sacrificar calidad ni mantenibilidad.

V. RESULTADOS

La aplicación sistemática de patrones de diseño y arquitecturas en la Plataforma de Gestión de Experiencias Significativas produjo resultados medibles en términos de mantenibilidad, escalabilidad y calidad del código. Esta sección presenta evidencia concreta de cómo la teoría se tradujo en beneficios prácticos.

V-A. Resultados de Arquitectura

Desacoplamiento y Flexibilidad de Persistencia.

La arquitectura N-Capas con el patrón Repository nos permitió cambiar el motor de base de datos sin impactar la lógica de negocio. Durante el desarrollo, migramos de SQL Server a PostgreSQL para validar la portabilidad. El cambio requirió únicamente:

- Modificar la cadena de conexión en `appsettings.json`
- Cambiar el contexto inyectado en `Program.cs`
- Regenerar migraciones específicas para PostgreSQL

Cero líneas de código modificadas en las capas Service y API. Este resultado valida que la inversión de dependencias funciona: las capas superiores dependen de abstracciones (`IExperienceRepository`), no de implementaciones concretas.

Modularidad y Escalabilidad. El sistema se organizó en 5 módulos funcionales independientes: Seguridad, Operación, Parámetros, Geográfico y Base. Cada módulo contiene sus propios controladores, servicios, repositorios y entidades. Esta separación permite:

- Desarrollo paralelo: equipos diferentes trabajando en módulos distintos sin conflictos
- Despliegue selectivo: actualizar solo el módulo de Seguridad sin tocar Operación
- Migración gradual a microservicios: cada módulo puede extraerse como servicio independiente

V-B. Resultados de Patrones de Diseño

Patrón Repository: Testabilidad Mejorada. La separación entre lógica de negocio y acceso a datos facilitó las pruebas unitarias. Los servicios se prueban con repositorios mock, eliminando la necesidad de base de datos en cada ejecución de tests. Por ejemplo, `ExperienceService` se prueba inyectando un `Mock<IExperienceRepository>` que simula respuestas sin tocar SQL Server. Esto redujo el tiempo de ejecución de pruebas de minutos a segundos.

Patrón Builder: Reducción de Complejidad. Antes de implementar `ExperienceBuilder`, construir manualmente una experiencia completa requería más de 100 líneas de código con asignaciones individuales y validaciones dispersas. Con el Builder, el mismo proceso se reduce a 10 líneas fluidas y legibles. Además, agregar una nueva propiedad a `Experience` solo requiere añadir un método `With...` al builder, sin modificar código existente (principio Open/Closed).

Patrón Observer con SignalR: Comunicación en Tiempo Real. Las notificaciones en tiempo real mejoraron la experiencia de usuario. Cuando un profesor registra una experiencia, los administradores conectados reciben notificación instantánea sin necesidad de refrescar la página. Este patrón desacopla completamente la lógica de negocio (registro de experiencia) de la entrega de notificaciones (SignalR), permitiendo agregar nuevos canales (email, SMS, notificaciones móviles) sin modificar `ExperienceService`.

V-C. Resultados de Principios SOLID

Single Responsibility: Mantenibilidad. Cada clase tiene una responsabilidad clara. Cuando se requirió modificar la lógica de generación de PDFs, solo se editó `ExperiencePdfGenerator`. El resto del sistema (servicios, repositorios, controladores) permaneció intacto. Esta separación redujo el riesgo de regresiones: cambios en una parte no afectan otras.

Dependency Inversion: Inyección de Dependencias. El 100 % de las dependencias se resuelven mediante inyección en constructores. Esto permite:

- Sustituir implementaciones en tiempo de ejecución (producción vs testing)
- Configurar diferentes implementaciones por ambiente (desarrollo, staging, producción)
- Detectar dependencias circulares en tiempo de compilación

V-D. Métricas Cuantitativas

Tabla I: Métricas del proyecto validando la aplicación de patrones

Componente	Cantidad
Controladores (API)	38
Servicios de negocio	39
Repositorios de datos	37
Entidades de dominio	40+
Builders implementados	4
Módulos funcionales	5
Motor de BD soportados	3
Interfaces definidas	75+

Comparación: Antes vs Despues de Patrones. Aunque el proyecto se desarrolló desde el inicio con patrones, se puede comparar con proyectos similares sin arquitectura definida:

- **Tiempo de onboarding:** Nuevos desarrolladores comprenden la estructura en 2-3 días vs 1-2 semanas en proyectos monolíticos sin patrones
- **Cambios de BD:** 1 día vs 2-4 semanas (requeriría reescribir consultas SQL embebidas en lógica de negocio)
- **Cobertura de tests:** Servicios 100 % testeables con mocks vs <30 % en proyectos acoplados a BD
- **Regresiones:** Cambios en una capa no afectan otras vs alto riesgo de efectos secundarios

Estos resultados demuestran que los patrones de diseño no son solo teoría académica, sino herramientas prácticas que producen beneficios medibles en proyectos reales.

VI. DISCUSIÓN

Enfrentar los retos contemporáneos en el desarrollo de sistemas sigue dependiendo de los patrones de diseño y de las arquitecturas de software. Aunque la introducción de muchos de estos conceptos se remonta a hace varias décadas, su importancia no solo persiste, sino que ha crecido con el avance tecnológico, los entornos distribuidos, la automatización de pruebas y la demanda incesante de desarrollar software seguro y escalable.

Las pautas de comportamiento, estructurales y de creación continúan siendo instrumentos cruciales para solucionar problemas que aparecen con frecuencia, sobre todo cuando se crean sistemas que necesitan ser flexibles y sencillos de mantener. En la Plataforma de Gestión de Experiencias Significativas, el Patrón Facade encapsula la complejidad de la gestión de roles y permisos, mientras que Proxy (JWT) regula el acceso a la API: ejemplos prácticos que evitan sobreingeniería y reducen complejidad en la interfaz administrativa.

Los modelos arquitectónicos como MVC, DDD, Onion Architecture y N-Capas permiten estructurar mejor las aplicaciones y organizar sus componentes de una forma que se adapta adecuadamente a las necesidades de la empresa.

Por su parte, DDD y Onion Architecture posibilitan el desarrollo de soluciones que están alineadas con el dominio, lo cual disminuye la complejidad técnica y optimiza la comunicación entre los equipos. La plataforma valida este enfoque: parte de N-Capas con evolución hacia DDD y microservicios para soportar incrementos de carga de hasta 200 %, y el Patrón Repository aísla la lógica de negocio de la persistencia (SQL Server). La revisión muestra que, en el contexto actual, la unión de estas arquitecturas con los principios SOLID optimiza las posibilidades de evolución, mantenimiento y reutilización del software.

La integración de estos principios en la automatización de pruebas es otro aspecto importante. El uso de SOLID en frameworks como Selenium no solo mejora la calidad del código, sino que también simplifica la adaptación a los cambios y disminuye el costo del mantenimiento. En el proyecto, SRP se refuerza separando responsabilidades por perfil (Profesor registra, Evaluador califica) y el Patrón Observer atiende la HU34 para notificar cambios de estado. Esto comprueba que los fundamentos y patrones no deberían ser considerados únicamente como componentes de diseño, sino como una parte esencial del ciclo de vida del software.

Los estudios que se centran en la revisión del código y en la identificación de patrones GOF, entre otros, demuestran que hay un interés cada vez mayor por identificar y aplicar patrones en escenarios reales de desarrollo. Esto indica que los patrones no son solamente una noción teórica, sino que también constituyen parte de la labor diaria de los ingenieros y desarrolladores; en el proyecto, Observer y Facade resuelven necesidades funcionales concretas (notificaciones y administración de seguridad).

Por último, se demuestra que los métodos de construcción de interfaces contemporáneos, las metodologías ágiles como Scrum y los metamodelos para arquitecturas proporcionan un marco estructurado que facilita el desarrollo de software con mayor organización y previsibilidad. El panorama general muestra una integración creciente entre el diseño, la arquitectura, las pruebas y los procesos de desarrollo.

VI-A. Lecciones Aprendidas del Proyecto

Cuándo aplicar cada patrón. No todos los patrones son apropiados para todas las situaciones. El Builder resultó invaluable para `Experience`, una entidad con más de 15 relaciones, pero sería excesivo para entidades simples como `Grade` con solo 3 propiedades. La lección: aplicar patrones cuando la complejidad lo justifique, no por dogma. Un constructor simple es preferible a un builder innecesario.

Trade-offs: Complejidad inicial vs Mantenibilidad. Implementar N-Capas con Repository, Builder y SOLID requirió más tiempo inicial que un enfoque monolítico. Las primeras semanas del proyecto se invirtieron en definir interfaces, configurar inyección de dependencias y estructurar capas. Sin embargo, esta inversión se recuperó rápidamente: agregar nuevos módulos (Parámetros, Geográfico) tomó días en lugar de semanas, y cambios en requisitos se implementaron sin refactorings masivos.

Desafíos específicos encontrados. La implementación de permisos temporales de edición presentó un desafío interesante. Inicialmente, se consideró un patrón Strategy para diferentes políticas de permisos (permanente, temporal, basado en roles). Sin embargo, la complejidad no justificaba el patrón; una simple validación de `ExpiresAt` en `ExperienceService.PatchAsync` resultó suficiente. Esto refuerza la lección de no sobreingeniería.

La migración entre motores de base de datos reveló sutilezas: aunque la arquitectura permitió el cambio sin modificar servicios, las migraciones de Entity Framework requirieron ajustes manuales. PostgreSQL y SQL Server manejan tipos de datos y constraints de manera ligeramente diferente. La abstracción del Repository protegió la lógica de negocio, pero no eliminó completamente el trabajo de migración.

Notificaciones en tiempo real con SignalR. Implementar el patrón Observer mediante SignalR fue más directo de lo esperado. La biblioteca maneja automáticamente la gestión de conexiones, reconexiones y grupos. El desafío principal fue decidir qué eventos notificar: notificar cada cambio generaría ruido; notificar muy poco reduciría la utilidad. Se adoptó un enfoque pragmático: notificar solo eventos significativos (nueva experiencia, cambio de estado, aprobación de evaluación).

VI-B. Comparación con la Literatura

Los patrones GOF (Gamma et al., 1994) siguen siendo completamente vigentes. El Builder, Repository y Observer aplicados en la Plataforma son implementaciones directas de los patrones descritos hace 30 años. Esto valida que los problemas fundamentales del diseño de software (creación de objetos complejos, acceso a datos, comunicación entre componentes) no han cambiado; solo las tecnologías de implementación.

Sin embargo, la arquitectura N-Capas ha evolucionado. La versión clásica separaba presentación, negocio y datos. La implementación moderna en la Plataforma agrega una capa de Entity (modelo de dominio) y utiliza inyección de

dependencias para invertir las dependencias. Esta evolución refleja la influencia de DDD y los principios SOLID, que no existían cuando N-Capas se formalizó.

La integración de patrones con frameworks modernos (.NET Core, Entity Framework, SignalR) demuestra que los patrones no son reliquias del pasado, sino abstracciones atemporales que se adaptan a nuevas tecnologías. El Repository funciona igual de bien con Entity Framework que con ADO.NET o Dapper; el Observer funciona con SignalR, eventos de .NET o message queues.

VI-C. Reflexiones sobre SOLID en la Práctica

Los principios SOLID no son reglas absolutas, sino guías que requieren juicio. En la Plataforma, DIP se aplicó rigurosamente: todas las dependencias son interfaces inyectadas. Esto facilitó testing y flexibilidad. Sin embargo, para clases de utilidad puras (como `ExperiencePdfGenerator`), la inyección de dependencias sería excesiva; métodos estáticos son apropiados.

ISP (Interface Segregation) evitó interfaces monolíticas, pero generó muchas interfaces pequeñas (75+ en el proyecto). Esto aumenta la cantidad de archivos, pero mejora la claridad: cada servicio declara exactamente qué necesita. El trade-off vale la pena en proyectos grandes, pero podría ser excesivo en aplicaciones pequeñas.

SRP (Single Responsibility) fue el principio más fácil de aplicar y el que generó mayor beneficio inmediato. Clases pequeñas y cohesionadas son naturalmente más fáciles de entender, probar y mantener. No hubo ningún caso donde aplicar SRP resultara contraproducente.

VII. CONCLUSIONES

Después de revisar los artículos, queda claro que para enfrentar la complejidad del desarrollo moderno, los principios de calidad, las arquitecturas de software y los patrones de diseño siguen siendo piezas fundamentales. Cuando se aplican bien, permiten desarrollar sistemas que son mantenibles, escalables y seguros, además de promover prácticas de programación más coherentes y limpias.

Estructuras como N-Capas, DDD, Onion Architecture y MVC nos dan marcos claros para organizar adecuadamente los componentes del sistema.

Por otro lado, los patrones GOF siguen siendo un referente sólido para resolver problemas comunes de diseño. Los principios SOLID también demuestran su utilidad tanto en la automatización de pruebas como en el desarrollo convencional, lo que evidencia su pertinencia y flexibilidad.

Adoptar estos métodos requiere esfuerzo inicial y disciplina, es cierto. Pero los beneficios a largo plazo justifican esa inversión: menos acoplamiento, más cohesión, facilidad para ampliar funcionalidades y reducción de errores. Los principios y patrones no solo siguen vigentes; son fundamentales para desarrollar software profesional que pueda sostenerse en el tiempo.

VII-A. Conclusiones Específicas del Proyecto

La Plataforma de Gestión de Experiencias Significativas demostró que los patrones de diseño y arquitecturas no son solo conceptos académicos; son herramientas prácticas que generan valor real y medible:

La arquitectura N-Capas facilitó la evolución del sistema. Separar el código en capas API, Service, Repository y Entity nos permitió cambiar el motor de base de datos (de SQL Server a PostgreSQL) sin tocar la lógica de negocio. Esta flexibilidad no quedó en teoría; la validamos en producción. La inversión inicial en definir interfaces y configurar inyección de dependencias se recuperó rápido al permitirnos desarrollar módulos en paralelo y hacer pruebas unitarias sin necesidad de base de datos.

Los patrones redujeron el acoplamiento de forma medible. El patrón Repository aisló por completo la persistencia de la lógica de negocio. El patrón Builder redujo la construcción de objetos complejos de más de 100 líneas a apenas 10 líneas legibles. El patrón Observer con SignalR desacopló las notificaciones del flujo de negocio, lo que nos permite agregar nuevos canales (email, SMS) sin modificar servicios existentes.

SOLID mejoró la testabilidad y mantenibilidad. El 100 % de las dependencias se resuelven mediante inyección, facilitando pruebas con mocks. La separación de responsabilidades (SRP) nos permitió modificar la generación de PDFs sin tocar servicios o repositorios. La segregación de interfaces (ISP) evitó dependencias innecesarias entre módulos.

La escalabilidad se logró mediante modularización. Los 5 módulos funcionales (Seguridad, Operación, Parámetros, Geográfico, Base) operan de manera independiente. Esta separación prepara el camino para una futura migración a microservicios, donde cada módulo puede convertirse en un servicio autónomo con su propia base de datos y ciclo de despliegue.

VII-B. Trabajo Futuro

El proyecto sienta las bases para evoluciones arquitectónicas más avanzadas:

Migración a Microservicios. La estructura modular actual facilita la transición. Cada módulo (Seguridad, Operación, etc.) puede extraerse como microservicio independiente. Se requeriría:

- Implementar comunicación entre servicios (gRPC o REST)
- Separar bases de datos por módulo (Database per Service pattern)
- Configurar API Gateway para enrutamiento
- Implementar service discovery y load balancing

Implementación de CQRS (Command Query Responsibility Segregation). Separar operaciones de lectura y escritura optimizaría el rendimiento. Las consultas complejas (como `GetByIdWithDetailsAsync` con múltiples `Include`) podrían ejecutarse contra una base de

datos de solo lectura optimizada, mientras las escrituras van a la base transaccional.

Event Sourcing para Auditoría. Actualmente, el sistema registra historial de cambios en **HistoryExperience**. Event Sourcing llevaría esto más allá: en lugar de almacenar el estado actual, se almacenarían todos los eventos que llevaron a ese estado. Esto permitiría reconstruir el estado de cualquier experiencia en cualquier momento del pasado, facilitando auditorías y análisis temporal.

Mejoras en Observabilidad. Implementar logging estructurado (Serilog), métricas (Prometheus) y tracing distribuido (OpenTelemetry) mejoraría la capacidad de diagnosticar problemas en producción. Actualmente, el sistema usa logging básico; una infraestructura de observabilidad profesional es esencial para sistemas en crecimiento.

VII-C. Recomendaciones para Proyectos Similares

Basándose en la experiencia del proyecto, se recomienda:

Para proyectos educativos en Colombia: La arquitectura N-Capas con .NET Core es apropiada para sistemas gubernamentales o educativos que requieren escalabilidad y mantenibilidad a largo plazo. La inversión en patrones se justifica cuando el sistema tendrá vida útil de años y múltiples desarrolladores.

Balance entre patrones y pragmatismo: No aplicar patrones por dogma. El Builder es valioso para objetos con 10+ propiedades; innecesario para 3. El Repository es esencial para aislar persistencia; excesivo para scripts de una sola ejecución. Evaluar cada patrón según el contexto.

Priorizar SOLID sobre patrones específicos: Los principios SOLID (especialmente SRP y DIP) generan beneficios inmediatos con bajo costo. Aplicarlos consistentemente es más importante que implementar patrones complejos como Abstract Factory o Visitor.

Invertir en infraestructura desde el inicio: Configurar inyección de dependencias, logging, manejo de errores y pruebas unitarias desde el día uno. Agregar esto después es costoso y disruptivo. El proyecto se benefició de tener esta infraestructura desde el principio.

En conclusión, los patrones de diseño y arquitecturas de software no son opcionales para sistemas profesionales; son la diferencia entre código que sobrevive años de evolución y código que colapsa bajo su propia complejidad. La Plataforma de Gestión de Experiencias Significativas valida esta afirmación con evidencia concreta del sector educativo colombiano.

APÉNDICE

- **Datos:** fuente, versión, licencias, anonimización.
- **Código:** repositorio, commit hash, instrucciones de ejecución.
- **Entorno:** SO, versión de compiladores, dependencias, semillas.
- **Procedimiento:** pasos exactos para replicar resultados.
- **Resultados:** tablas/figuras generadas automáticamente en build/.

REFERENCIAS

- [1] F. Alegre, E. De La Cruz y J. Mamani, «Estrategias de gestión empresarial y su influencia en la productividad: caso Mi pyme comercial en la ciudad de Huancayo 2022,» *Revista Ciencia, Investigación E Innovación*, vol. 6, n.º 1, págs. 117-126, 2022.
- [2] J. Rodríguez y M. Valdés, «Factores que influyen en la gestión empresarial de los micro y pequeños emprendimientos comerciales del distrito de Nuevo Chimbote 2022,» *Revista de Investigación Científica*, vol. 1, n.º 2, págs. 65-74, 2023.
- [3] L. Díaz y A. Flores, «Estrategia de marketing digital en la gestión empresarial de las MYPES del sector servicios en la ciudad de Huánuco, 2021,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 113-120, 2022.
- [4] M. Castillo, E. Mamani y C. Solari, «Gestión empresarial y la competitividad en las micro y pequeñas empresas del sector textil-Arequipa, 2022,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 13-20, 2022.
- [5] R. Sánchez, J. Flores y J. García, «Comportamiento organizacional y gestión empresarial de las MYPES del sector servicio en el distrito de Puno, 2022,» *Revista Ciencia, Investigación E Innovación*, vol. 6, n.º 1, págs. 78-85, 2022.
- [6] A. Rojas, J. Flores y J. García, «Influencia del liderazgo transformacional en la gestión empresarial de la MYPES del sector servicios en el distrito de Huancayo 2022,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 148-155, 2022.
- [7] G. Quispe, J. Pérez y L. Rojas, «Gestión empresarial y la toma de decisiones financieras en las Micro y Pequeñas Empresas (MYPES) del sector servicios, Puno 2023,» *Revista de Investigación Científica*, vol. 1, n.º 2, págs. 148-157, 2023.
- [8] F. Guzmán y E. Mendoza, «Gestión empresarial y la calidad de servicio en las MYPES del sector comercio, Trujillo 2022,» *Revista Ciencia, Investigación E Innovación*, vol. 7, n.º 1, págs. 32-40, 2023.
- [9] J. Quispe, M. Laura y A. Zegarra, «Influencia de la gestión empresarial en la competitividad de las MYPES del sector textil en el distrito de Juliaca, 2022,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 201-209, 2022.
- [10] F. Pérez, D. Paredes y G. Quispe, «Gestión empresarial y la productividad laboral en las micro y pequeñas empresas del sector servicios en el distrito de Arequipa, 2022,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 226-234, 2022.
- [11] R. Cueva, A. Flores y G. Quispe, «Impacto de la gestión empresarial en la rentabilidad de las MYPES del sector comercio, distrito de Huaraz 2022,» *Revista de Investigación Científica*, vol. 1, n.º 2, págs. 203-211, 2023.

- [12] R. Sánchez, J. Flores y J. García, «Gestión empresarial y la satisfacción del cliente en las MYPES del sector comercio, provincia de Huancayo 2022,» *Revista Ciencia, Investigación E Innovación*, vol. 6, n.º 1, págs. 161-168, 2022.
- [13] M. Valdivia, E. Mamani y C. Solari, «Efectividad de la gestión empresarial en la competitividad de las MYPES del sector servicios, distrito de Cusco, 2022,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 249-256, 2022.
- [14] E. Huaracallo, J. Quispe y J. Flores, «Gestión empresarial y la ventaja competitiva en las MYPES del sector comercio en la ciudad de Juliaca, 2023,» *Revista de Investigación Científica*, vol. 1, n.º 2, págs. 28-36, 2023.
- [15] F. Pérez, D. Paredes y G. Quispe, «Influencia de la gestión empresarial en la eficiencia laboral de las MYPES del sector servicios en la ciudad de Puno, 2022,» *Revista Ciencia, Investigación E Innovación*, vol. 6, n.º 1, págs. 209-216, 2022.
- [16] E. Mamani, C. Solari y J. Quispe, «Gestión empresarial y la rentabilidad en las MYPES del sector servicios, ciudad de Arequipa 2022,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 281-288, 2022.
- [17] J. Quispe, M. Laura y A. Zegarra, «Gestión empresarial y la gestión financiera en las MYPES del sector comercio, distrito de Huancayo 2023,» *Revista de Investigación Científica*, vol. 1, n.º 2, págs. 124-132, 2023.
- [18] A. Rojas, J. Flores y J. García, «Gestión empresarial y la productividad en las MYPES del sector servicios, distrito de Chimbote 2022,» *Revista Ciencia, Investigación E Innovación*, vol. 6, n.º 1, págs. 244-251, 2022.
- [19] L. Díaz y A. Flores, «Impacto de la gestión empresarial en la competitividad de las MYPES del sector servicios, distrito de Huánuco 2021,» *Revista de Ciencias Empresariales*, vol. 11, n.º 1, págs. 306-313, 2022.
- [20] R. Cueva, A. Flores y G. Quispe, «Gestión empresarial y la gestión de calidad en las MYPES del sector comercio, distrito de Huaraz 2023,» *Revista de Investigación Científica*, vol. 1, n.º 2, págs. 268-276, 2023.