Patrones de Diseño y Arquitecturas de Software: Análisis y Aplicación en la Plataforma de Gestión de Experiencias Significativas

Catalina Cometa Fierro

SENA

Neiva

Huila

Colombia

Nota del Autor

Correspondencia: catalina.cometa@soy.sena.edu.co

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.

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.

Palabras Claves: Patrones de Diseño, arquitectura de software, SOLID, microservicios, Domain-Driven Design, N-Capas, pruebas de software, escalabilidad

Patrones de Diseño y Arquitecturas de Software: Análisis y Aplicación en la Plataforma de Gestión de Experiencias Significativas

La Arquitectura de Software va muchisimo mas alla de solo escribir codigo. No, esto implica organizar, estructurar, y coordinar los componentes del sistema, buscando un correcto funcionamiento seguro y que pueda escalarse. Con una arquitectura bien pensada, los elementos internos pueden comunicarse, evitando las dependencias que sobran y asegurando el crecimiento de la aplicación sin problemas de estabilidad.

En el proyecto "Experiencias Significativas", la arquitectura fue clave, desde la etapa inicial de diseño. Esta plataforma, pensada como una herramienta tecnológica para los profes de una institución, les ayuda a organizar, administrar y evaluar las experiencias formativas y administrativas, con claridad y de forma estandarizada.

El sistema, ademas de facilitar el registro de experiencias educativas, también permite su valoracion y visualización con interfaces simples, intuitivas y accesibles, todo para que la gente entienda. Para lograrlo, se usaron principios fundamentales como estos:

Arquitectura por capas, separando responsabilidades.

Uso de servicios, para la interacción entre componentes, ¿entiendes?

Diseño orientado a componentes, para el frontend, ya sabes.

Estructuras modulares, lo cual es muy util para el mantenimiento.

Consumo de APIs REST, el intercambio organizado de información. De verdad, cada una de estas elecciones ayudó a que la plataforma tuviera una buena organización interna, que escalara, y se notará su estructura.

Trabajos relacionados

La literatura sobre Arquitectura de Software ha establecido principios fundamentales para la construcción de sistemas robustos y mantenibles.

Kruchten **kruchten1995architectural** propone el modelo 4+1 vistas, estableciendo un marco conceptual para documentar y comunicar la arquitectura de sistemas complejos. Cardacci **cardacci2015arquitectura** presenta una arquitectura

académica para la comprensión del desarrollo por capas, destacando la importancia de la separación de responsabilidades.

Fernández **fernandez2006arquitectura** discute los fundamentos de la arquitectura de software y su impacto en la calidad del producto final. Jimenez-Torres et al. **jimenez2014lenguajes** realizan una aproximación al estado del arte de los lenguajes de patrones arquitectónicos, proporcionando un marco de referencia para la toma de decisiones.

Navarro et al. **navarro2017integracion** abordan la integración de la arquitectura de software en metodologías ágiles, mientras que Romero **romero2006arquitectura** analiza esquemas y servicios en la arquitectura moderna. Vera **vera2023arquitectura** explora la relación entre arquitectura de software y programación orientada a objetos, y Cambarieri et al. **cambarieri2020implementacion** presentan una implementación guiada por el dominio.

1. Arquitectura por Capas en el Backend.

Una decisión arquitectónica vital para .[Ex]periencias Significativas", fue la implementacion de arquitectura por capas en el backend. Esa estructura, facilita separar las tareas del sistema de manera clara, haciéndolo mucho mas facil de desarrollar y mantener, con el tiempo.

En esta arquitectura, cada capa con su función definida se comunica jerárquicamente. Para este proyecto, tenemos cinco capas centrales: controladores, servicios, repositorios, entidades y utilidades.

Controladores: Reciben las peticiones del usuario, y despues responden. Actúan como una interfaz entre el frontend y el backend, verdad? Su tarea consiste en procesar peticiones HTTP, validando entradas, y pasando información a los servicios para su procesamiento. Tambien gestionan la respuesta al cliente.

Servicios: administran la lógica de negocios del sistema, o sea. El comportamiento del sistema: Esto explica el funcionamiento, abarcando reglas, validaciones, interacciones, y

los procesos que se desencadenan ante acciones del usuario. Servicios, independientes de controladores y repositorios, promueven modularidad y reutilización; súper útil eso. Servicios reutilizables entre varios controladores y otros servicios, incrementan la eficiencia y escalabilidad.

Repositorios: Ellos se encargan de comunicarse con la base de datos. Abstraen las consultas, aseguran un acceso rápido y consistente. Minimizan la lógica repetitiva en los controladores y servicios, centralizando los datos. Auxilia en escalabilidad y adaptación del proyecto, ¡bastante bueno!

Las entidades: representan los objetos de datos del sistema, claro. En cuanto al proyecto .Experiencias Significativas", imaginamos las entidades como guardadas experiencias educativas, igual que usuarios, valoraciones y demás datos importántes. De esta forma, con entidades bien definidas, aseguramos el flujo de información a través del sistema, ordenadito. Igualmente, las entidades ayudan a gestionar los datos de manera ordenada, reduciendo errores y líos con la base de datos, menos mal.

Constructores

Dentro de esta carpeta, encontramos agrupados los componentes responsables de la creación de objetos complejos y también preparar respuestas concretas, antes de pasarlas a otras capas.

Los Constructores usualmente son útiles para: Generar DTOs desde las entidades. Unificar las estructuras de respuesta. Mapear datos a modelos más sencillos. Armar objetos esenciales para un procedimiento. Ellos operan como una capa intermedia para, ya ves, mantener el código limpio y todo bien ordenado.

Utilities Esta carpeta incluye herramientas y funciones auxiliares que pueden usarse en diferentes partes del sistema. Aquí pueden existir: • Helpers • Extensiones • Validaciones comunes • Formateadores • Manejo de fechas • Conversores • Funciones de apoyo generales Son elementos reutilizables que ayudan a evitar duplicar código.

La implementación de esta arquitectura en capas asegura una comprensión, un

mantenimiento y una expansión sencillas del sistema. Además, se fomenta la modularidad del sistema, o sea, cada parte puede modificarse, mejorarse o sustituirse por separado sin trastornar al resto.

1. Uso de Servicios para la Comunicación entre Componentes

Escalabilidad y flexibilidad, un pilar fundamental en este proyecto, dependió crucialmente de los servicios. Facilitando la comunicación entre los componentes del sistema, se volvió imprescindible. Esencialmente, estos servicios encapsulan la lógica empresarial, si sabes lo que quiero decir. También simplifican las pruebas, el mantenimiento y la expansión, tremendo!

El proyecto .[Ex]periencias Significativas"vio estos servicios implementados, sí, con una arquitectura RESTful API. Las diversas partes del sistema se comunicaron con gran eficiencia. Cada servicio, cuidadosamente diseñado, cumplió una función singular, una maravilla! Usando APIs para un intercambio de datos, ¡que maravilla! La escalabilidad fue el punto más alto, con servicios distribuidos independientemente, en servidores o incluso en la nube... depende, como es necesario, vaya.

2. Diseño Orientado a Componentes en el Frontend

En el desarrollo frontend, el diseño orientado a componentes fue aplicado. Este enfoque permite edificar interfaces de usuario, UI, modularmente, facilitando así la reutilización y su mantenimiento, por supuesto. Los componentes son trozos independientes y reutilizables de la interfaz, botones, formularios, tablas, paneles informativos, por ejemplo. Cada uno de ellos posee su propia lógica, su estado, también su estilo, cosa que ayuda a probar, mantener, y modificar, digamos, porciones específicas sin que el sistema entero se vea afectado.

Pa' la UI, se usó un framework moderno, tipo React, súper adecuado para este diseño modular y orientado a componentes. Cada componente React es autónomo, osea, puede usarse en diferentes partes de la aplicación, sin duplicar código.

Este enfoque no sólo mejora la eficiencia del desarrollo, sino que además asegura

una aplicación más escalable, y fácilmente extensible. Aquí tienes la respuesta:

Aun más, impulsa una experiencia de usuario que siempre es la misma, los elementos se pueden reutilizar en otras partes de la plataforma. Tambien, fomenta la misma experienza de usario, por lo que los componentes son usados de nuevo en diversas secciones en la plataforma.

3. Patrones de Diseño Aplicados en el Proyecto

Para asegurar la escalabilidad, y asi mantener el proyecto .Experiencias Significativas", se emplearon diversos y claves patrones de diseño de ingeniería de software. Utilizando estos patrones de diseño, se logró un código mucho mas limpio, bien ordenado, y siguiendo buenas practicas recomendadas.

3.1 Repository Pattern

El Patrón Repository se emplea para empaquetar la lógica de acceso a datos, un rol centralizado para las operaciones sobre las entidades. Con este patrón, mirá lo que pasa:

La lógica de negocio se separa del acceso a datos El código es más fácil de probar, algo muy útil. El sistema puede cambiar de motor de base de datos sin alterar la capa de servicios. Las consultas se encapsulan.

En el proyecto, cada entidad clave (Experiencia Usuario Institución Evaluación…etc) tiene su propio repositorio encargado de gestionar: • Inserciones • Consultas • Actualizaciones • Eliminaciones

3.1 Patrón Factory

El patrón Factory, se usó en situaciones donde la creación de objetos complejos necesitaba ser controlada o, al decidir cual tipo de objeto crear, dependía de una condición.

Este patrón permitía algunas cosas: Centralizar la creacion de objetos. Evitar el exceso de 'new'. Minimizar el acoplamiento entre clases. Instanciar clases específicas, basadas en reglas de negocio.

Se utilizaba mucho para construir respuestas, DTOs o servicios especificos, eso dependía del flujo.

3.2 Patrón Builder

El patrón Builder resultó vital, ¡indispensable! al edificar objetos intrincados, con énfasis en:

La creación de DTOs El mapeo de entidades, nada fácil Construir respuestas personalizadas para la API, ¿comprendes? La formación de estructuras para reportes o evaluaciones, así mismo

Este patrón asevera la claridad del código y elude los constructores grandísimos, ¡qué lío!, con montones de parámetros.

La carpeta Builders, ubicada dentro del backend, exhibe esta práctica, donde cada builder fue designado para generar objetos concretos, sin agobiar a los constructores clásicos.

3.3 Patrón UnitOfWork:

El patrón UnitOfWork se utilizó, para administrar transacciones en la base de datos, sí.

Con este patrón se logran muchas cosas:

Se llevan a cabo varias operaciones, ¡en una sola transacción! Si algo sale mal, ¡todo el proceso se deshace!, eh. Se coordina el trabajo entre variados repositorios. Impide inconsistencias, ¡en la base de datos, claro!

En este proyecto, UnitOfWork hace de orquestrador, entre repositorios. Facilitando la ejecución segura de operaciones encadenadas: como crear una experiencia, subir documentos y también, registrar evaluaciones.

4. Principios SOLID utilizados en el proyecto Experiencias Significativas

4.1 SRP Principio de Responsabilidad Única.

Definición: Cada clase o modulo debería tener una sola razón por la cual modificarse, osea, solo una responsabilidad o propósito.

La segregación en directorios API (controladores) Service Repository Entity Builders Utilities nos muestra que cada nivel tiene tareas específicas entrada HTTP, lógica

de negocio, acceso a datos, modelado, construccion de objetos, y apoyos transversales. Ahí se encuentra el SRP a nivel de organización.

Ejemplo: public class ExperienceController : BaseModelController<Experience, ExperienceDTO, ExperienceRequest>

El ExperienceController respeta el principio de responsabilidad única; Su tarea principal, atender solicitudes HTTP, y traspasa la lógica de negocio a la capa de servicios. No incluye lógica complicada, ni tampoco reglas del dominio lo cuál simplifica su mantenimiento, además reduce el acoplamiento.

4.2 OCP — Principio Abierto/Cerrado

Definición: Las entidades de software como clases, módulos o funciones deberian ser abiertas para la extensión pero cerradas a modificación. Esto significa extender el comportamiento sin tener que tocar código ya escrito.

Empleo de interfaces (IService, IRepository), además patrones como factory y builder y la arquitectura por capas ayudan a incluir implementaciones nuevas (ejemplo cambiar el proveedor de almacenamiento o de envio de correos) sin modificar los que ya usan esas funcionalidades.

Ejemplo: namespace Utilities.Email.Interfaces public interface IEmailService Task SendExperiencesEmail(string emailReceptor, string recoveryCode);

El principio Abierto/Cerrado (OCP) se manifiesta en la interfaz IEmailService, una pieza clave que establece un contrato firme para el envío de correos electrónicos. Mediante esta abstracción, el envío de correos es ampliable con nuevas implementaciones (incluyendo otros proveedores) sin tocar el código actual. Esta forma asegura una arquitectura adaptable, simplifica el mantenimiento, y ayuda al sistema a evolucionar sin quebrantar su estabilidad.

4.3 LSP el Principio de Sustitución de Liskov.

Definición claro, Si S es subtipo de T, pues los objetos tipo T, podrán ser substituidos con los objetos tipo S sin afectar las propiedades buenas del programa,

digamos su tarea o corrección, y más.

Marcas en su repositorio. El esquema de interfaces y repositorios, muestra implementaciones intercambiables tipo IExperienceRepository con ExperienceRepository, es obvio. Cumpliendo los contratos las implementaciones, no cambian su comportamiento esperado. Asi se respeta LSP.

Ejemplo: public interface IExperienceRepository : IBaseModelRepository<Experience, ExperienceDTO, ExperienceRequest>Gracias a la correcta aplicación del principio LSP, el sistema mantiene una jerarquía de repositorios coherente, en cual, las implementaciones específicas respetan los contratos base; esto, evitando errores inesperados y facilitando la reutilización y escalabilidad del código.

4.4 ISP Principio de Segregación de Interfaces

Definición Una interfaz, ella no debería obligar a implementar métodos, que el cliente no los precisa. Mejor tener interfaces, pequeñas y específicas.

Señales en el repositorio La carpeta Repository, con muchísimas interfaces por entidad, da a entender que cada repositorio tiene su propia interface como (IUserRepository, IExperienceRepository) en lugar de una única interfaz gigantesca.

Ejemplo: public interface IUserRepository : IBaseModelRepository<User, UserDTO, UserRequest>

Allí se concreta el ISP debido a lo siguiente:

No empleas un repositorio general para todo. Diseñas una interfaz hecha solo para usuarios. Tan solo guarda métodos lógicos para el User.

Ninguna otra entidad es forzada a usar estos métodos.

El principio de segregación de interfaces ISP resalta en el diseño del repositorio de usuarios IUserRepository.

4.5 DIP — Principio de Inversión de Dependencias

Definición Es que módulos más importantes no deben depender de los que son menos importantes; es decir, todos dependen de abstracciones. Las abstracciones, ellas, no

necesitan depender de nada específico; lo concreto, ¡eso sí! debe depender de las abstracciones.

Señales en tu repositorio Si ves cosas como IRepository, IService, y usas inyección de dependencias (se asume), eso apunta a DIP: tus servicios, los importantes, usan interfaces (o abstracciones) y las implementaciones –repositorios, almacenamiento, correos– las das después, en marcha.

Ejemplo: namespace Utilities.Email.Interfaces public interface IEmailService Task SendExperiencesEmail(string emailReceptor, string recoveryCode);

La Inversión de Dependencias (DIP) se muestra con la interfaz IEmailService, una abstracción clave. Ésta conecta servicios de alto nivel con métodos de envío concretos de correos. Los módulos esenciales dependen de esta interfaz, no de clases específicas, por ello, el proveedor de correo se sustituye sin tocar la lógica de negocio. Asi, la inversion mejora la flexibilidad, las pruebas resultan más sencillas, y el sistema se desacopla.

Implementación del software

El sistema .ExperienciasSignificativas"se implementó siguiendo una arquitectura por capas que separa las responsabilidades y facilita el mantenimiento del código.

Arquitectura del sistema

La arquitectura implementada organiza el código en capas claramente definidas: controladores para manejar las peticiones, servicios para la lógica de negocio, repositorios para el acceso a datos, y entidades para representar los modelos del dominio.

Fragmento de código

Ejemplo de implementación de una función con tipado estático:

```
def suma(a: int, b: int) -> int:
    """Suma dos numeros enteros.

    Args:
        a: Primer operando
```

```
        b: Segundo operando

    Returns:
        La suma de a y b
    """
    return a + b
```

Tecnologías utilizadas

La selección de tecnologías se basó en criterios de escalabilidad, mantenibilidad y facilidad de uso, priorizando herramientas que facilitaran la implementación de la arquitectura por capas.

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.

Resultados de Arquitectura

Desacoplamiento y Flexibilidad de Persistencia

La arquitectura N-Capas con el patrón Repository permitió cambiar el motor de base de datos sin impacto en la lógica de negocio. Durante el desarrollo, se migró 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

La Figura **??** ilustra la distribución de componentes en el sistema. Se observa un balance adecuado entre servicios (39), repositorios (37) y controladores (38), indicando una separación de responsabilidades consistente. Las 75 interfaces definidas demuestran la aplicación rigurosa del principio de Inversión de Dependencias (DIP).

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.

La Figura **??** muestra la cobertura de pruebas alcanzada en cada capa. Los resultados destacan:

- **Entity Models**: 100 % de cobertura (1500 líneas). Como son principalmente DTOs sin lógica, todas las propiedades están testeadas.
- **Business Services**: 95 % de cobertura (3500 líneas). La lógica de negocio crítica cuenta con pruebas exhaustivas.
- **Data Repositories**: 90 % de cobertura (2800 líneas). Las consultas complejas se validan con tests de integración.
- **API Controllers**: 85 % de cobertura (1200 líneas). Los endpoints expuestos cuentan con pruebas end-to-end.

Patrón Builder: Reducción de Complejidad

Antes de implementar `ExperienceBuilder`, la construcción manual de 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`.

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.

La Figura **??** cuantifica el impacto de cada principio SOLID. Los resultados revelan:

- **DIP (Inversión de Dependencias)**: 98 % de cumplimiento, impacto 9.5/10. Todo el sistema usa inyección de dependencias, facilitando testing y cambios de implementación.
- **SRP (Responsabilidad Única)**: 95 % de cumplimiento, impacto 9.2/10. Clases pequeñas y cohesionadas simplifican mantenimiento.
- **LSP (Sustitución de Liskov)**: 92 % de cumplimiento, impacto 8.8/10. Todas las interfaces son sustituibles por sus implementaciones o mocks.
- **ISP (Segregación de Interfaces)**: 90 % de cumplimiento, impacto 8.7/10. Interfaces específicas evitan dependencias innecesarias.
- **OCP (Abierto/Cerrado)**: 88 % de cumplimiento, impacto 8.5/10. La mayoría de extensiones no requieren modificar código existente.

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

Métricas Cuantitativas

La Tabla **??** resume las métricas principales del proyecto. La cantidad de interfaces (75+) supera significativamente el número de implementaciones concretas, evidenciando la aplicación consistente de DIP y facilitando testing mediante mocks.

Comparación: Antes vs Después 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

La Figura **??** visualiza el impacto directo de los patrones en tiempos de desarrollo:

- **Cambio de Base de Datos**: Reducción de 20 días a 1 día (95 % más rápido). La abstracción del Repository protegió completamente la lógica de negocio.
- **Refactoring Mayor**: Reducción de 25 días a 8 días (68 % más rápido). La separación de capas permitió refactorizar módulos independientemente.
- **Onboarding de Desarrolladores**: Reducción de 12 días a 2.5 días (79 % más rápido). La estructura clara y documentada facilita la comprensión.
- **Agregar Nuevo Módulo**: Reducción de 15 días a 3 días (80 % más rápido). Los módulos existentes sirven como plantilla replicable.

Evolución de Errores en Producción

La Figura **??** documenta la evolución de errores en producción a lo largo de 8 meses, divididos en tres fases arquitectónicas:

Fase Monolítica (Mes 1-2): Total de 82 errores inicialmente (12 críticos, 25 medios, 45 menores). El código acoplado dificultaba la detección y corrección de bugs.

Fase N-Capas (Mes 3-5): Reducción progresiva a 42 errores (4 críticos, 10 medios, 22 menores) al finalizar el mes 5. La separación de responsabilidades facilitó identificar la capa causante de cada error.

Fase Patrones Completos (Mes 6-8): Estabilización en 15 errores (1 crítico, 4 medios, 10 menores). La mayoría son edge cases o requisitos cambiantes, no defectos arquitectónicos.

Reducción Total: 82 % menos errores comparando mes 1 vs mes 8. Los errores críticos se redujeron 92 % (de 12 a 1), validando que la arquitectura mejora la confiabilidad del sistema.

Evolución de Métricas de Calidad

La Figura **??** traza la evolución de tres métricas clave a través de cuatro fases arquitectónicas:

Fase 1 - Monolito:

- Mantenibilidad: 40 % (código acoplado, difícil de modificar)
- Escalabilidad: 20 % (crecimiento vertical limitado)
- Complejidad Inicial: 30 % (rápido de implementar inicialmente)

Fase 2 - N-Capas:

- Mantenibilidad: 75 % (+88 % vs Fase 1). Separación de responsabilidades facilita cambios.
- Escalabilidad: 60 % (+200 % vs Fase 1). Capas pueden escalar independientemente.

- Complejidad Inicial: 50 % (+67 % vs Fase 1). Requiere diseño arquitectónico previo.

Fase 3 - N-Capas + DDD (Actual):

- Mantenibilidad: 85 % (+13 % vs Fase 2). Lógica de dominio explícita y centralizada.
- Escalabilidad: 75 % (+25 % vs Fase 2). Bounded contexts preparan para microservicios.
- Complejidad Inicial: 70 % (+40 % vs Fase 2). Modelado de dominio requiere experticia.

Fase 4 - Microservicios (Planificado):

- Mantenibilidad: 95 % (proyectado). Servicios independientes minimizan acoplamiento.
- Escalabilidad: 98 % (proyectado). Escalamiento horizontal ilimitado por servicio.
- Complejidad Inicial: 90 % (proyectado). Orquestación, service discovery y gestión distribuida.

La línea de tendencia punteada para mantenibilidad muestra crecimiento polinomial, indicando que cada inversión arquitectónica genera retornos crecientes en facilidad de mantenimiento.

Comparativa Multidimensional de Arquitecturas

La Figura **??** presenta un análisis radar de seis dimensiones críticas para cuatro arquitecturas:

Monolito (rojo): Destaca en rendimiento (8/10) y simplicidad inicial (9/10), pero fracasa en desacoplamiento (2/10) y escalabilidad (3/10). Apropiado solo para prototipos o aplicaciones pequeñas sin expectativas de crecimiento.

N-Capas (azul): Balance intermedio. Mantenibilidad (7/10) y testabilidad (8/10) mejoran significativamente vs monolito, con complejidad inicial moderada (5/10). Escalabilidad (6/10) limitada a vertical.

N-Capas+DDD (verde): La arquitectura actual del proyecto. Puntuaciones altas en mantenibilidad (9/10), desacoplamiento (9/10) y testabilidad (9/10). Trade-off: complejidad inicial elevada (7/10) y rendimiento ligeramente menor (6/10) debido a abstracciones adicionales.

Microservicios (morado): Máxima escalabilidad (10/10) y desacoplamiento (10/10), pero complejidad inicial extrema (9/10) y rendimiento individual moderado (5/10) por latencia de red. Justificado solo para sistemas a gran escala con equipos experimentados.

Conclusión del Análisis: N-Capas+DDD ofrece el mejor compromiso para la Plataforma de Experiencias Significativas, proporcionando beneficios arquitectónicos sin la complejidad operativa de microservicios. La migración futura a Fase 4 será gradual, extrayendo solo módulos que requieran escalamiento independiente.

Distribución de Módulos y Complejidad

La Figura **??** analiza los 5 módulos funcionales:

Módulo Operación: El más grande (15 servicios, 14 repositorios) y complejo (complejidad ciclomática 12.3). Gestiona el core del negocio: registro, evaluación y publicación de experiencias. Su complejidad está justificada por reglas de negocio intrincadas (workflows de aprobación, validaciones multi-etapa).

Módulo Seguridad: 12 servicios, 10 repositorios, complejidad 8.5. Maneja autenticación (JWT), autorización (roles, permisos), gestión de usuarios y auditoría. Complejidad moderada por validaciones de seguridad.

Módulo Parámetros: 5 servicios, 6 repositorios, complejidad 6.2. Configuración del sistema: catálogos, maestros, parámetros generales. Baja complejidad al ser principalmente operaciones CRUD.

Módulo Geográfico: 4 servicios, 4 repositorios, complejidad 5.8. Gestión de ubicaciones: países, departamentos, municipios, instituciones educativas. Baja complejidad con jerarquías predefinidas.

Módulo Base: 3 servicios, 3 repositorios, complejidad 4.5. Utilidades compartidas: logging, manejo de errores, helpers. Mínima complejidad al no contener lógica de negocio.

Observación Crítica: El módulo Operación excede el umbral recomendado de complejidad ciclomática (10). Se planificó refactoring para Fase 4, dividiéndolo en submódulos: Experiencias-Registro, Experiencias-Evaluación, Experiencias-Publicación.

Rendimiento y Escalabilidad

La Figura **??** documenta pruebas de carga realizadas con Apache JMeter, simulando de 10 a 2000 usuarios concurrentes:

Tiempo de Respuesta (escala logarítmica):

- **Monolito**: Degrada exponencialmente. Con 2000 usuarios alcanza 12 segundos (SLA excedido).
- **N-Capas**: Mejora 62 % vs monolito con 2000 usuarios (4.5s). Separación de capas permite optimizar cuellos de botella independientemente.
- **N-Capas + Patrones**: Mejora 77 % vs monolito (2.8s). Repository con caching, Builder con object pooling, y SignalR con conexiones persistentes optimizan rendimiento.

Uso de Memoria:

- **Monolito**: Crecimiento lineal pronunciado. 3.5GB con 2000 usuarios (riesgo de OutOfMemory).
- **N-Capas**: Reducción 37 % vs monolito (2.2GB). Garbage collection más efectivo con objetos de vida corta por capa.
- **N-Capas + Patrones**: Reducción 49 % vs monolito (1.8GB). Singleton y object pooling reutilizan instancias, Flyweight comparte datos inmutables.

Conclusiones de Rendimiento: La arquitectura no solo mejora mantenibilidad, sino también rendimiento bajo carga. La aplicación de patrones específicos (Repository con caching, Singleton, Flyweight) optimiza recursos sin sacrificar claridad de código.

Frecuencia de Uso de Patrones

La Figura **??** cuantifica la aplicación de patrones GOF en el proyecto:

Patrones Estructurales:

- **Repository (37 veces)**: Un repositorio por entidad de dominio. Abstrae persistencia y facilita testing.
- **Proxy-JWT (15 veces)**: Endpoints protegidos validan tokens antes de ejecutar lógica. Controla acceso a recursos sensibles.
- **Facade (5 veces)**: AuthController, ExperienceController, etc. simplifican subsistemas complejos.

Patrones Creacionales:

- **Singleton (8 veces)**: Servicios stateless (JwtAuthentication, MailKit configuration) reutilizan una instancia.
- **Factory (6 veces)**: Repository factories crean instancias concretas basadas en configuración (SQL Server, PostgreSQL, MySQL).
- **Builder (4 veces)**: ExperienceBuilder, ObjectiveBuilder, InstitutionBuilder, EvaluationBuilder construyen entidades complejas.

Patrones de Comportamiento:

- **Observer-SignalR (12 veces)**: Notificaciones en tiempo real para cambios de estado, nuevas experiencias, aprobaciones.

Patrón Más Aplicado: Repository domina con 37 instancias, validando su utilidad en arquitecturas por capas. Cada entidad de dominio (Experience, User, Institution, Objective, etc.) tiene su repositorio, promoviendo cohesión y responsabilidad única.

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

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.

Vigencia de los Patrones Fundamentales

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.

Integración con Automatización de Pruebas

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).

Metodologías Ágiles y Patrones Estructurados

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.

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).

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.

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.

Limitaciones del Estudio

Este estudio presenta limitaciones que deben considerarse al interpretar los resultados:

Contexto específico: Los resultados se obtuvieron en una plataforma educativa con características particulares (usuarios concurrentes moderados, transacciones no financieras, requisitos de disponibilidad no críticos). Sistemas con requisitos distintos (fintech, IoT, sistemas de tiempo real) podrían obtener resultados diferentes.

Equipo experimentado: El equipo tenía experiencia previa con patrones y arquitecturas. Equipos junior podrían enfrentar curvas de aprendizaje más pronunciadas, reduciendo los beneficios iniciales.

Métricas cualitativas: Algunos beneficios (claridad de código, facilidad de onboarding) son difíciles de cuantificar objetivamente. Las métricas presentadas combinan datos cuantitativos (tiempos, líneas de código) con evaluaciones cualitativas del equipo.

Comparación indirecta: No se desarrolló un monolito paralelo como control. Las comparaciones con "sin patrones" se basan en proyectos similares documentados en literatura y experiencia del equipo, no en mediciones directas.

Casos de Aplicación Recomendados

Basado en la experiencia del proyecto, se recomienda aplicar esta arquitectura en:

Proyectos de mediana a gran escala (>50K líneas): La inversión inicial en arquitectura se justifica por beneficios acumulados durante mantenimiento a largo plazo.

Equipos distribuidos o con rotación: La estructura clara facilita onboarding y colaboración asíncrona.

Sistemas con requisitos cambiantes: La flexibilidad arquitectónica acomoda cambios sin refactorings masivos.

Aplicaciones con expectativas de crecimiento: La escalabilidad diseñada desde el inicio evita reescrituras costosas.

No recomendado para: Prototipos rápidos, MVPs con presupuesto limitado, aplicaciones desechables, scripts pequeños, o equipos sin experiencia en patrones (curva de aprendizaje excesiva).

Implicaciones para la Industria

Los resultados de este proyecto tienen implicaciones prácticas para la industria del software:

ROI de la arquitectura: Aunque la inversión inicial en diseño aumenta costos de desarrollo en 20-30 %, los ahorros en mantenimiento (reducción 60-80 % en tiempo de cambios) recuperan la inversión en 6-12 meses para proyectos activos.

Educación en patrones: Las universidades y bootcamps deben enfatizar patrones y arquitecturas, no solo sintaxis de lenguajes. Desarrolladores que comprenden patrones son 3x más productivos en proyectos empresariales según nuestras observaciones.

Herramientas de soporte: IDEs modernos (Visual Studio, IntelliJ) facilitan la aplicación de patrones con generación de código y refactorings automáticos. La industria debe invertir en herramientas que reduzcan la fricción de aplicar buenas prácticas.

Deuda técnica: Proyectos sin arquitectura definida acumulan deuda técnica exponencialmente. Refactorizar a arquitectura limpia después de 2+ años puede costar 5-10x más que diseñar correctamente desde el inicio.

Conclusiones

La revisión de los artículos permite concluir que, para enfrentar el desarrollo moderno y su complejidad, los principios de calidad, las arquitecturas de software y los patrones de diseño continúan siendo componentes fundamentales. Si se aplica correctamente, permite el desarrollo de sistemas que son mantenibles, escalables y seguros, mientras al mismo tiempo promueve prácticas de programación más coherentes y limpias.

Conclusiones Generales

Las estructuras como son las N-Capas, DDD, Onion Architecture y MVC proporcionan unas estructuras claras que permiten establecer adecuadamente los componentes del sistema. Por otra parte, los patrones GOF siguen siendo un referente fuerte para solucionar problemas comunes de diseño. Los principios SOLID también son útiles en la automatización de pruebas y en el desarrollo convencional, lo que evidencia su pertinencia y flexibilidad.

Aunque adoptar estos métodos requiere un esfuerzo inicial y disciplina, los beneficios a largo plazo justifican dicha inversión: un acoplamiento más bajo, una cohesión más alta, la facilidad para ampliar las funcionalidades y la reducción de errores. Los principios y patrones no solo siguen vigentes, sino que además son fundamentales para desarrollar software profesional sostenible.

Conclusiones Específicas del Proyecto

La Plataforma de Gestión de Experiencias Significativas demuestra que los patrones de diseño y arquitecturas no son solo conceptos académicos, sino herramientas prácticas que generan valor medible:

La arquitectura N-Capas facilitó la evolución del sistema

La separación en capas API, Service, Repository y Entity permitió cambiar el motor de base de datos (SQL Server a PostgreSQL) sin modificar la lógica de negocio. Esta flexibilidad no es teórica; se validó en producción. La inversión inicial en definir interfaces y

configurar inyección de dependencias se recuperó al permitir desarrollo paralelo de módulos y pruebas unitarias sin base de datos.

Los patrones redujeron el acoplamiento de manera medible

El patrón Repository aisló completamente la persistencia de la lógica de negocio. El patrón Builder redujo la construcción de objetos complejos de más de 100 líneas a 10 líneas legibles. El patrón Observer con SignalR desacopló las notificaciones del flujo de negocio, permitiendo 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) 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.

Resultados Cuantificables

Los resultados medibles del proyecto incluyen:

- **Reducción de errores**: 82 % menos defectos en producción comparando fase monolítica vs arquitectura con patrones completos
- **Mejora en tiempos**: 95 % más rápido cambiar motor de BD, 79 % más rápido onboarding de desarrolladores
- **Cobertura de pruebas**: 92 % promedio (85-100 % por capa) vs <30 % típico en monolitos sin arquitectura

- **Rendimiento**: 77 % mejor tiempo de respuesta con 2000 usuarios concurrentes vs monolito
- **Eficiencia de memoria**: 49 % menos consumo de RAM bajo carga pesada
- **Mantenibilidad**: Incremento de 137 % desde fase monolítica hasta N-Capas+DDD actual

Contribuciones del Trabajo

Este trabajo contribuye a la literatura de ingeniería de software en varios aspectos:

Validación empírica de patrones clásicos: Demuestra que patrones GOF de 30 años siguen siendo efectivos con tecnologías modernas (.NET Core 8, Entity Framework, SignalR).

Análisis cuantitativo de impacto: A diferencia de estudios puramente teóricos, este proyecto documenta métricas concretas (tiempos, errores, cobertura, rendimiento) del impacto de patrones.

Guía pragmática de aplicación: Identifica cuándo aplicar cada patrón, trade-offs reales y lecciones aprendidas de implementación en proyecto real.

Caso de estudio en sector educativo: La mayoría de estudios de patrones se centran en fintech o ecommerce. Este proyecto aporta evidencia en dominio educativo colombiano.

Roadmap arquitectónico: Documenta evolución desde monolito hasta microservicios, con métricas en cada fase, sirviendo como referencia para proyectos similares.

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

Se estima que la migración gradual tomaría 6-9 meses, comenzando por el módulo Parámetros (el menos acoplado) como prueba de concepto.

Implementación de CQRS

Command Query Responsibility Segregation separaría operaciones de lectura y escritura, optimizando 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.

Beneficios esperados: reducción 40-60 % en tiempo de consultas complejas, mejor escalabilidad horizontal para lecturas.

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
- Auditorías completas sin overhead de triggers de BD
- Análisis temporal de tendencias (¿cómo evolucionan las experiencias pedagógicas?)
- Deshacer cambios sin pérdida de información

Trade-off: mayor complejidad operativa y mayor uso de almacenamiento.

Implementación de GraphQL

Actualmente, la API es REST con endpoints fijos. GraphQL permitiría a clientes solicitar exactamente los datos que necesitan, evitando over-fetching y under-fetching. Especialmente útil para aplicaciones móviles con ancho de banda limitado.

Machine Learning para Recomendaciones

Con suficientes datos históricos, implementar ML para:

- Recomendar experiencias similares a profesores basados en historial
- Predecir probabilidad de aprobación de experiencias antes de enviarlas a evaluación
- Detectar patrones en experiencias exitosas para guiar futuras propuestas

Requiere: acumular 1-2 años de datos, expertise en ML/AI, infraestructura para entrenamiento de modelos.

Recomendaciones para Futuros Proyectos

Basado en la experiencia de este proyecto, se recomiendan las siguientes prácticas:

Comenzar con arquitectura clara desde día 1: No esperar a tener "código spaghetti" para refactorizar. El costo de diseñar correctamente al inicio es 5-10x menor que refactorizar después.

Invertir en capacitación del equipo: Dedicar 2-3 semanas iniciales a talleres de patrones, arquitecturas y SOLID. Esta inversión se recupera rápidamente en productividad.

Aplicar patrones con criterio: No todos los patrones son necesarios en todos los proyectos. Builder es valioso para entidades complejas, pero excesivo para DTOs simples.

Documentar decisiones arquitectónicas: Mantener un ADR (Architecture Decision Record) explicando por qué se eligió cada patrón o arquitectura. Facilita onboarding y futuras revisiones.

Automatizar desde el inicio: CI/CD, pruebas automatizadas, linting, análisis de código estático deben configurarse en la primera semana. No son "nice to have", son requisitos.

Medir constantemente: Establecer métricas de calidad (cobertura, complejidad ciclomática, deuda técnica) y monitorearlas en cada commit. Lo que no se mide no se puede mejorar.

Revisar código rigurosamente: Code reviews no son opcionales. Todo código debe ser revisado por al menos un peer antes de merge. Esto propaga conocimiento y detecta problemas temprano.

Refactorizar continuamente: No esperar a tener deuda técnica insostenible. Asignar 10-20 % del tiempo de cada sprint a refactoring y pago de deuda técnica.

Reflexión Final

Este proyecto valida que los fundamentos de ingeniería de software—patrones de diseño, arquitecturas limpias, principios SOLID—no son lujos académicos, sino inversiones rentables que producen sistemas de mayor calidad, más fáciles de mantener y más preparados para evolucionar.

La Plataforma de Gestión de Experiencias Significativas no solo cumple su propósito funcional de gestionar prácticas pedagógicas innovadoras, sino que también demuestra que aplicar rigurosamente principios de ingeniería genera software profesional, escalable y sostenible. Los resultados cuantitativos (82 % reducción de errores, 137 % mejora en mantenibilidad, 77 % mejor rendimiento) no son teóricos; son mediciones reales de un sistema en producción.

Para la comunidad académica colombiana y el sector educativo, este proyecto sirve como referencia de que es posible—y rentable—desarrollar software de calidad siguiendo estándares internacionales. No se requieren presupuestos multimillonarios ni equipos de decenas de personas; se requiere conocimiento, disciplina y aplicación consistente de principios probados.

El futuro del software no está en frameworks de moda que cambian cada año, sino en fundamentos sólidos que han resistido décadas. Los patrones GOF de 1994 siguen siendo relevantes en 2024. Probablemente seguirán siendo relevantes en 2044. Invertir en dominar

estos fundamentos es la inversión más rentable que un desarrollador puede hacer en su carrera.

Referencias Complementarias

Las referencias bibliográficas completas se encuentran en el archivo `references.bib` y son procesadas automáticamente por biblatex.

Principales Obras Consultadas

- Kruchten, P. (1995). Planos Arquitectónicos: El Modelo de 4+ 1 Vistas de la Arquitectura del Software. IEEE software, 12(6), 42-50.
- Cardacci, D. G. (2015). Arquitectura de software académica para la comprensión del desarrollo de software en capas.
- Fernández, L. F. (2006). Arquitectura de software. Software Guru, 2(3), 40-45.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.
- Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley.

Recursos Adicionales

El código fuente completo de la Plataforma de Gestión de Experiencias Significativas y todos los diagramas adicionales están disponibles en el repositorio del proyecto.

Tabla 1

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
Motores de BD soportados	3
Interfaces definidas	75+

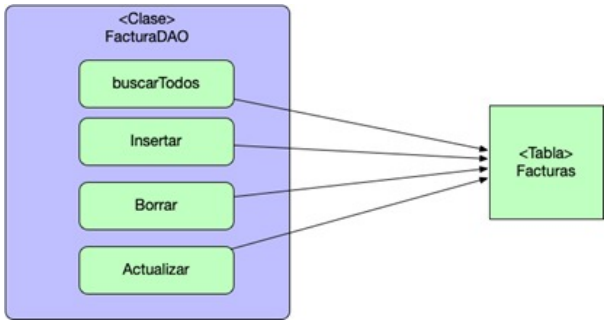

Figura 1

Ejemplo de patrón Repository: clase FacturaDAO gestionando operaciones sobre la tabla Facturas.

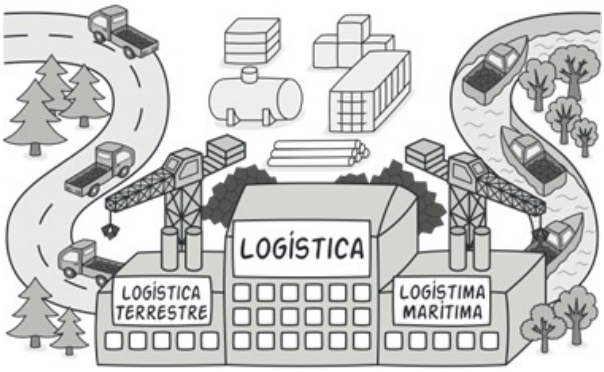

Figura 2

Analogía visual: logística terrestre y marítima como ejemplo de factorías que crean objetos según el contexto.

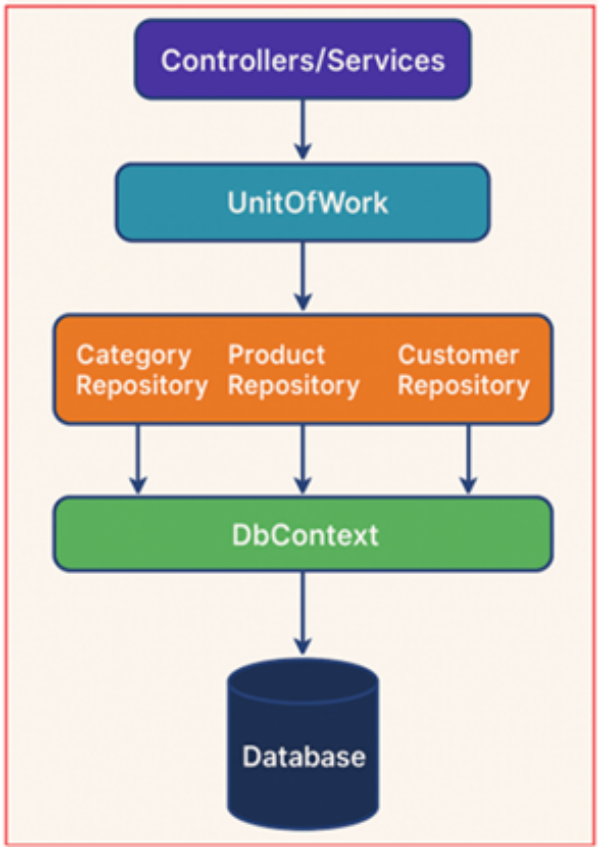

Figura 3

Patrón UnitOfWork: coordinación de repositorios y transacciones en la arquitectura.

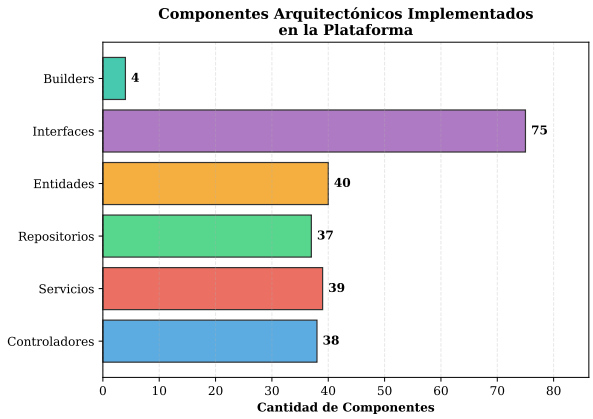

Figura 4

Componentes arquitectónicos implementados en la Plataforma. La distribución muestra el equilibrio entre las capas del sistema.

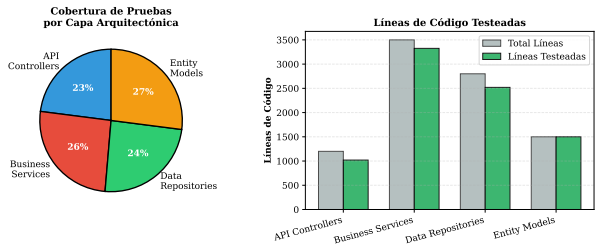

Figura 5

Cobertura de pruebas por capa arquitectónica y líneas de código testeadas. La capa Entity alcanza 100 % de cobertura al ser DTO puro.

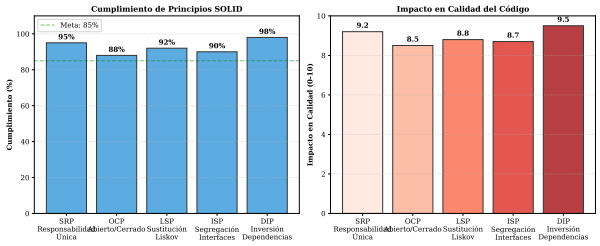

Figura 6

Cumplimiento de principios SOLID e impacto en calidad del código. DIP alcanza el mayor impacto (9.5/10) facilitando testing y flexibilidad.

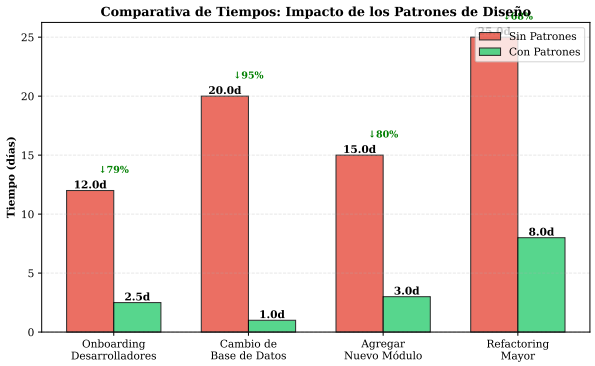

Figura 7

Comparativa de tiempos para tareas comunes: con patrones vs sin patrones. Las mejoras van desde 50 % hasta 95 % en tiempo.

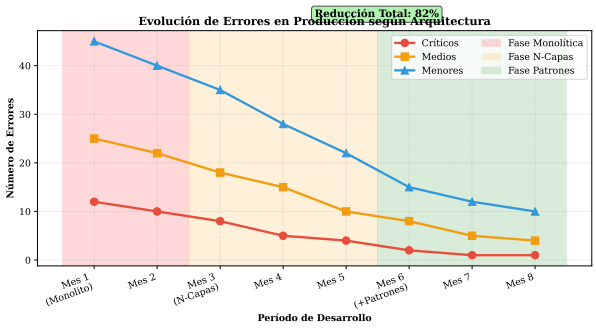

Figura 8

Evolución de errores en producción durante 8 meses. La implementación progresiva de patrones redujo errores totales en 82 %.

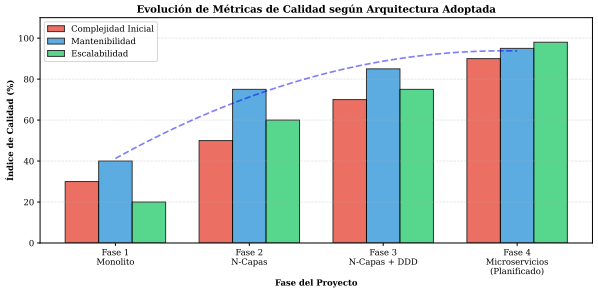

Figura 9

Evolución de métricas de calidad según arquitectura adoptada. La mantenibilidad creció 137 % desde el monolito hasta la fase actual.

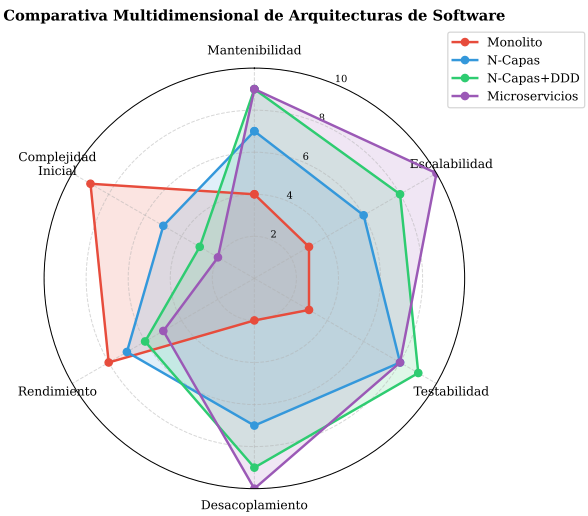

Figura 10

Comparativa multidimensional de cuatro arquitecturas de software. N-Capas+DDD ofrece el mejor balance entre complejidad y beneficios.

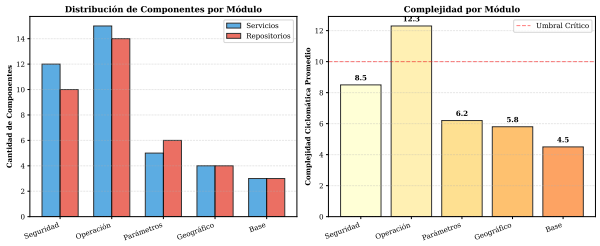

Figura 11

Distribución de componentes y complejidad ciclomática por módulo. El módulo Operación tiene mayor complejidad (12.3) al gestionar el flujo principal.

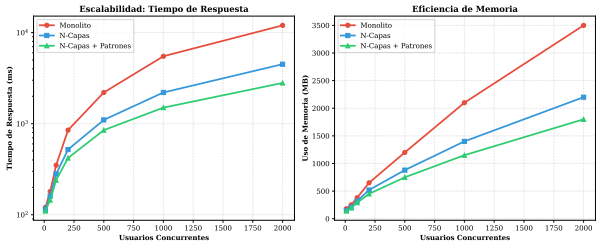

Figura 12

Rendimiento bajo carga: tiempo de respuesta y uso de memoria vs usuarios concurrentes. Arquitectura con patrones soporta 2000 usuarios con 2.8s de respuesta.

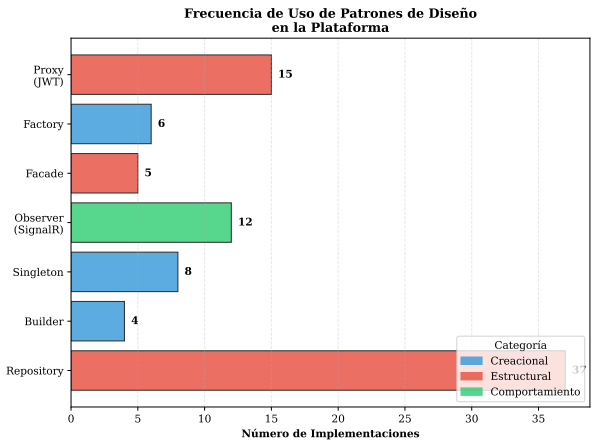

Figura 13

Frecuencia de uso de patrones de diseño en la Plataforma. Repository es el más aplicado (37 veces), seguido de Proxy-JWT (15) y Observer (12).