Instituto Politécnico Nacional

Escuela Superior de Cómputo

Web Client and Backend Development Frameworks.

Reporte de Investigación: Arquitectura de Cebolla (Clean Architecture)

Profesor: M. en C. José Asunción Enríquez Zárate

Alumno: Segura Gutierrez Obed

osegurag2100@alumno.ipn.mx

7CM1

7 de octubre de 2025

Índice

Índice de figuras

1. Introducción

La Arquitectura de Cebolla (Onion Architecture), también conocida como Arquitectura Limpia (Clean Architecture), es un patrón arquitectónico propuesto por Jeffrey Palermo en 2008 y posteriormente refinado por Robert C. Martin (Uncle Bob) en su libro Çlean Architecture"publicado en 2017. Este modelo arquitectónico busca crear sistemas de software altamente mantenibles, testeables y desacoplados de frameworks externos, representando una evolución natural de los principios SOLID aplicados a nivel arquitectónico.

La característica fundamental de esta arquitectura es su estructura en capas concéntricas, donde las dependencias apuntan siempre hacia el centro, es decir, hacia el dominio de la aplicación. Esto garantiza que la lógica de negocio permanezca independiente de frameworks, bases de datos, interfaces de usuario y cualquier otro agente externo. El nombre Çebolla"proviene precisamente de esta disposición en capas que recuerda a las capas de una cebolla, donde cada capa exterior depende de las capas interiores, pero nunca al revés.

1.1. Contexto Histórico

Antes de la aparición de la Arquitectura de Cebolla, el desarrollo de software empresarial se basaba principalmente en arquitecturas en capas tradicionales (Layered Architecture), donde la capa de presentación dependía de la capa de lógica de negocio, y esta a su vez dependía de la capa de acceso a datos. Este enfoque generaba varios problemas: fuerte acoplamiento con tecnologías específicas, dificultad para realizar pruebas unitarias, y sistemas rígidos difíciles de mantener y evolucionar.

El Domain-Driven Design (DDD), propuesto por Eric Evans en 2003, estableció las bases conceptuales al enfatizar la importancia del dominio de negocio. Posteriormente, la Arquitectura Hexagonal de Alistair Cockburn (también conocida como Ports and Adapters) introdujo la idea de invertir las dependencias para proteger el núcleo de la aplicación. Jeffrey Palermo sintetizó estas ideas en 2008 con su propuesta de Onion Architecture, que Robert C. Martin expandió y popularizó bajo el término Clean Architecture.

1.2. Motivación y Problemática

El desarrollo de software empresarial enfrenta constantemente desafíos relacionados con la mantenibilidad, escalabilidad y adaptabilidad a cambios tecnológicos. Las arquitecturas tradicionales en capas suelen crear un fuerte acoplamiento entre la lógica de negocio y la infraestructura, generando los siguientes problemas:

- **Dependencia Tecnológica:** La lógica de negocio queda atada a frameworks, bases de datos y tecnologías específicas, dificultando migraciones o actualizaciones
- **Dificultad para Testear:** Las pruebas unitarias requieren infraestructura completa (bases de datos, servidores, etc.), ralentizando el desarrollo
- **Rigidez ante Cambios:** Modificaciones en la infraestructura o presentación pueden requerir cambios en la lógica de negocio
- **Violación de Principios SOLID:** Especialmente el Principio de Inversión de Dependencias y el Principio de Responsabilidad Única
- **Acoplamiento Temporal:** Los componentes están sincronizados de manera que cambios en uno afectan a todos los demás

La Arquitectura de Cebolla surge como respuesta a estos problemas, proporcionando un modelo donde:

- La lógica de negocio es completamente independiente de la infraestructura
- Las pruebas unitarias son sencillas de implementar y ejecutar rápidamente
- Los cambios tecnológicos no afectan el núcleo del sistema
- El código es más legible, mantenible y evolucionable
- Se respetan los principios SOLID a nivel arquitectónico

1.3. Objetivos del Reporte

Este documento tiene como objetivos principales:

- Explicar en profundidad los conceptos teóricos fundamentales de la Arquitectura de Cebolla
- Describir exhaustivamente cada una de las capas que la componen y sus interrelaciones
- Analizar los principios arquitectónicos que sustentan este patrón
- Examinar las ventajas estratégicas y desventajas prácticas de este enfoque
- Comparar esta arquitectura con otros patrones arquitectónicos similares
- Proporcionar criterios para decidir cuándo es apropiado adoptar este patrón

2. Fundamentos Teóricos

La Arquitectura de Cebolla se fundamenta en principios sólidos de ingeniería de software que han demostrado su efectividad en la construcción de sistemas complejos y duraderos.

2.1. Principios SOLID

Los principios SOLID, acuñados por Robert C. Martin, constituyen la base teórica de la Arquitectura de Cebolla. Cada principio se aplica a nivel arquitectónico:

2.1.1. Single Responsibility Principle (SRP)

Cada capa de la arquitectura tiene una única responsabilidad bien definida. El dominio se encarga exclusivamente de las reglas de negocio, la capa de aplicación coordina casos de uso, la infraestructura maneja detalles técnicos, y la presentación gestiona la interacción con el usuario. Esta separación clara previene el acoplamiento y facilita el mantenimiento.

2.1.2. Open/Closed Principle (OCP)

La arquitectura está abierta a extensión pero cerrada a modificación. Nuevas funcionalidades se agregan mediante nuevas implementaciones de interfaces existentes, sin modificar el código del núcleo. Por ejemplo, se puede agregar un nuevo repositorio para una base de datos diferente sin tocar la lógica de negocio.

2.1.3. Liskov Substitution Principle (LSP)

Las implementaciones concretas en las capas externas pueden sustituirse por otras sin afectar el comportamiento del sistema. Un repositorio SQL puede reemplazarse por uno NoSQL siempre que ambos implementen la misma interfaz del dominio.

2.1.4. Interface Segregation Principle (ISP)

Las interfaces son específicas y cohesivas, evitando interfaces "gordas"que obliguen a implementar métodos innecesarios. Cada repositorio o servicio define únicamente los métodos relevantes para su propósito.

2.1.5. Dependency Inversion Principle (DIP)

Este es el principio fundamental de la Arquitectura de Cebolla. Los módulos de alto nivel (dominio) no dependen de módulos de bajo nivel (infraestructura), sino que ambos dependen de abstracciones. Las capas internas definen interfaces que las capas externas implementan, invirtiendo completamente la dirección de las dependencias tradicionales.

2.2. Regla de Dependencia

La regla más importante y estricta de la Arquitectura de Cebolla establece que las dependencias del código fuente solo pueden apuntar hacia adentro, nunca hacia afuera. Esto significa:

- Nada en un círculo interno puede saber absolutamente nada acerca de algo en un círculo externo
- El nombre de algo declarado en un círculo externo no debe mencionarse en el código de un círculo interno
- Esto incluye funciones, clases, variables, o cualquier otra entidad de software
- Los datos que cruzan las fronteras deben ser simples estructuras de datos o DTOs

Esta regla garantiza que el núcleo de la aplicación permanezca puro y libre de contaminación por detalles de implementación externa. Si la capa de dominio necesita persistencia, define una interfaz de repositorio (abstracción), y la capa de infraestructura proporciona la implementación concreta.

2.3. Separación de Responsabilidades (Separation of Concerns)

La Arquitectura de Cebolla lleva el principio de separación de responsabilidades al extremo, dividiendo el sistema en capas con fronteras claramente definidas:

- **Responsabilidades de Negocio:** Concentradas en el núcleo del dominio, completamente independientes de tecnología
- **Responsabilidades de Aplicación:** Coordinación de flujos de trabajo y orquestación de operaciones del dominio
- **Responsabilidades de Infraestructura:** Detalles técnicos de implementación, comunicación con sistemas externos
- **Responsabilidades de Presentación:** Interacción con usuarios o sistemas consumidores

Cada frontera representa un punto de desacoplamiento donde se pueden introducir abstracciones, facilitando pruebas, mantenimiento y evolución independiente de cada capa.

3. Estructura de Capas

La Arquitectura de Cebolla organiza el código en capas concéntricas, donde cada capa tiene responsabilidades específicas y bien definidas. La visualización como una cebolla ayuda a comprender que para llegar al núcleo (dominio), se deben atravesar las capas externas, pero el núcleo permanece completamente aislado e independiente.

3.1. Capa de Dominio (Core Domain)

Esta es la capa más interna y representa el corazón de la aplicación. Es el círculo más protegido y no tiene dependencias de ninguna otra capa o tecnología externa.

3.1.1. Componentes del Dominio

Entidades (Entities): Son objetos de negocio con identidad única que persiste a lo largo del tiempo. No son simples estructuras de datos, sino objetos que encapsulan reglas de negocio fundamentales. Una entidad Cliente, por ejemplo, no solo almacena datos del cliente, sino que también valida reglas como .el email debe ser único.o "la edad debe ser mayor de 18 años para ciertos productos".

Value Objects: Representan conceptos del dominio sin identidad propia. Son inmutables y se identifican por sus valores, no por un identificador. Ejemplos incluyen Dinero, Dirección, o RangoFecha. Dos objetos Dirección con los mismos valores son intercambiables y equivalentes.

Agregados (Aggregates): Son grupos de entidades y value objects que se tratan como una unidad para efectos de consistencia de datos. Cada agregado tiene una raíz (aggregate root) que es la única entidad a través de la cual se puede acceder a los objetos internos del agregado.

Domain Services: Operaciones de negocio que no pertenecen naturalmente a ninguna entidad específica. Por ejemplo, una operación de "Transferencia Bancariaïnvolucra múltiples cuentas y no pertenece a ninguna cuenta individual.

Domain Events: Representan hechos significativos que ocurren en el dominio. Por ejemplo, "PedidoConfirmado.o ÇlienteRegistrado". Permiten comunicación desacoplada entre diferentes partes del sistema.

Interfaces de Repositorio: Contratos que definen cómo se persisten y recuperan los agregados. Crucialmente, se definen en el dominio pero se implementan en la infraestructura, ejemplificando la inversión de dependencias.

3.1.2. Características Fundamentales

- **Cero Dependencias Externas:** No referencia ningún framework, librería de infraestructura, o componente de capa superior
- **Lógica de Negocio Pura:** Contiene únicamente reglas que reflejan el conocimiento del dominio del problema
- **Independencia Tecnológica:** No conoce nada sobre bases de datos, APIs, interfaces de usuario, o protocolos de comunicación
- **Testabilidad Completa:** Puede probarse exhaustivamente sin necesidad de bases de datos, servicios web, o cualquier infraestructura
- **Lenguaje Ubicuo:** Utiliza el vocabulario del dominio del negocio, facilitando la comunicación con expertos del dominio

3.2. Capa de Aplicación (Application Services)

Esta capa rodea al dominio y define los casos de uso del sistema, es decir, las operaciones específicas que la aplicación puede realizar. Actúa como coordinadora entre el mundo exterior y el dominio.

3.2.1. Componentes de la Capa de Aplicación

Casos de Uso (Use Cases): Representan las operaciones completas que un actor puede realizar en el sistema. Cada caso de uso es una transacción completa desde el punto de vista del usuario. Por ejemplo, RegistrarNuevoCliente", "ProcesarPedido", o "GenerarReporte".

DTOs (Data Transfer Objects): Objetos simples diseñados para transferir datos entre capas. No contienen lógica de negocio, solo propiedades y validaciones de formato básicas. Sirven como contratos de entrada y salida para los casos de uso.

Application Services: Orquestan el flujo de operaciones, invocando múltiples entidades del dominio, repositorios, y servicios externos. No contienen lógica de negocio propia, solo coordinación.

Mappers: Componentes responsables de transformar entre entidades del dominio y DTOs. Mantienen la separación entre el modelo de dominio interno y los modelos expuestos al exterior.

Validadores de Aplicación: Validan los datos de entrada antes de invocar el dominio. Estas validaciones son diferentes a las del dominio: aquí se valida formato, presencia de datos requeridos, etc., mientras que en el dominio se validan reglas de negocio.

3.2.2. Responsabilidades Principales

- **Coordinación de Operaciones:** Orquesta llamadas a múltiples componentes del dominio para completar un caso de uso
- **Gestión de Transacciones:** Define los límites transaccionales de las operaciones
- **Traducción de Datos:** Convierte entre el modelo externo (DTOs) y el modelo interno (entidades del dominio)
- **Autorización:** Verifica que el usuario tenga permisos para ejecutar la operación solicitada
- **Invocación de Servicios Externos:** Comunica con servicios de infraestructura mediante interfaces

3.3. Capa de Infraestructura

Esta capa contiene todas las implementaciones concretas de las abstracciones definidas en las capas internas. Es la capa que realmente "hace el trabajo sucio"de comunicarse con el mundo exterior.

3.3.1. Componentes de Infraestructura

Implementaciones de Repositorios: Clases concretas que implementan las interfaces de repositorio del dominio. Utilizan tecnologías específicas como Entity Framework, Hibernate, o acceso directo a SQL.

Contextos de Base de Datos: Configuración y administración de conexiones a bases de datos. Mapeo objeto-relacional (ORM), migraciones, y optimizaciones de consultas.

Clientes de Servicios Externos: Implementaciones para comunicarse con APIs de terceros, servicios de mensajería, colas, o cualquier sistema externo.

Implementaciones de Logging: Sistemas concretos de registro de eventos (Log4j, Serilog, etc.).

Gestión de Archivos: Operaciones de lectura/escritura en sistemas de archivos o almacenamiento en la nube.

Configuración: Lectura y procesamiento de archivos de configuración, variables de entorno, o sistemas de configuración centralizada.

3.3.2. Características de la Infraestructura

- **Dependencia de Tecnologías Específicas:** Esta capa puede y debe usar frameworks y librerías externas
- **Implementa Abstracciones:** Cumple con los contratos definidos por las capas internas
- **Sustituible:** Puede reemplazarse completamente sin afectar las capas internas
- **Optimizable:** Puede optimizarse para rendimiento sin afectar la lógica de negocio

3.4. Capa de Presentación

Es la capa más externa y maneja toda la interacción con el mundo exterior, ya sean usuarios humanos o sistemas consumidores.

3.4.1. Componentes de Presentación

Controladores/Endpoints: Manejan las peticiones entrantes (HTTP, gRPC, GraphQL, etc.), validan entrada básica, invocan casos de uso, y formatean respuestas.

ViewModels: Objetos específicamente diseñados para la presentación, conteniendo exactamente los datos que una vista necesita mostrar.

Middleware: Componentes que procesan las peticiones antes y después de llegar a los controladores (autenticación, logging, manejo de errores, etc.).

Validadores de Entrada: Validaciones específicas del protocolo de comunicación (formatos HTTP, límites de tamaño, etc.).

3.4.2. Responsabilidades de la Presentación

- **Traducción de Protocolos:** Convierte entre el protocolo de comunicación (HTTP, WebSockets, etc.) y las operaciones de aplicación
- **Formateo de Respuestas:** Serializa datos en el formato apropiado (JSON, XML, HTML, etc.)
- **Manejo de Errores:** Traduce excepciones internas en mensajes apropiados para el consumidor
- **Autenticación:** Verifica la identidad del usuario o sistema consumidor

4. Flujo de Control y Comunicación entre Capas

Comprender cómo fluye el control a través de las capas es crucial para implementar correctamente la Arquitectura de Cebolla.

4.1. Flujo de una Petición

El flujo típico de una petición sigue este patrón:

1. **Entrada en Presentación:** Una petición HTTP llega a un controlador
2. **Validación Superficial:** El controlador valida formato y datos básicos
3. **Invocación del Caso de Uso:** El controlador crea un DTO de entrada e invoca el caso de uso apropiado
4. **Validación de Negocio:** El caso de uso valida reglas de aplicación
5. **Creación/Recuperación de Entidades:** El caso de uso obtiene entidades del dominio mediante repositorios
6. **Ejecución de Lógica de Negocio:** Las entidades ejecutan sus reglas de negocio
7. **Persistencia:** Las entidades modificadas se persisten mediante repositorios
8. **Mapeo a DTO de Salida:** El resultado se convierte en un DTO apropiado
9. **Respuesta:** El controlador formatea y devuelve la respuesta

4.2. Cruce de Fronteras

Cuando los datos cruzan fronteras entre capas, deben transformarse. Nunca se pasan entidades del dominio directamente a capas externas. Este aislamiento permite:

- Evolución independiente del modelo de dominio
- Optimización de la representación de datos para cada capa
- Prevención de exposición accidental de detalles internos
- Facilitación de versionado de APIs

4.3. Inyección de Dependencias

La inversión de dependencias se implementa técnicamente mediante inyección de dependencias. Un contenedor IoC (Inversion of Control) resuelve las dependencias en tiempo de ejecución:

- Las capas internas definen interfaces (qué necesitan)
- Las capas externas implementan esas interfaces (cómo se hace)
- El contenedor IoC conecta las implementaciones con las interfaces
- El código del dominio solo conoce interfaces, nunca implementaciones concretas

5. Comparación con Otras Arquitecturas

Es importante comprender cómo la Arquitectura de Cebolla se relaciona con otros patrones arquitectónicos.

5.1. Arquitectura en Capas Tradicional

La arquitectura en capas tradicional organiza el código en capas horizontales: Presentación → Lógica de Negocio → Acceso a Datos. Las dependencias fluyen hacia abajo, creando acoplamiento fuerte con la base de datos.

Diferencias clave:

- En capas tradicionales, la lógica de negocio depende del acceso a datos
- En Onion, el dominio no depende de nada; la infraestructura depende del dominio
- Capas tradicionales dificultan las pruebas; Onion las facilita
- Cambiar de base de datos en capas tradicionales afecta todo el sistema; en Onion solo afecta la infraestructura

5.2. Arquitectura Hexagonal (Ports and Adapters)

La Arquitectura Hexagonal, propuesta por Alistair Cockburn, es conceptualmente muy similar a la Onion. Ambas buscan aislar el núcleo de la aplicación de detalles externos mediante abstracciones.

Relación y diferencias:

- Ambas invierten las dependencias hacia el dominio
- Hexagonal usa términos "puertosz .ªdaptadores"; Onion habla de capas concéntricas
- Hexagonal enfatiza puertos de entrada y salida; Onion enfatiza la jerarquía de capas
- Son complementarias y pueden considerarse diferentes visualizaciones del mismo concepto

5.3. Clean Architecture de Robert C. Martin

Clean Architecture es una generalización y expansión de Onion Architecture. Martin sintetiza varios patrones (Onion, Hexagonal, DCI, BCE) bajo un marco conceptual unificado.

Relación:

- Clean Architecture incluye los conceptos de Onion
- Añade capas adicionales como Ïnterface Adapters"
- Enfatiza más explícitamente la regla de dependencia
- En la práctica, implementar Onion correctamente es implementar Clean Architecture

5.4. Domain-Driven Design (DDD)

DDD no es una arquitectura sino una metodología de diseño que se complementa perfectamente con Onion:

- DDD define qué elementos deben estar en el dominio (entidades, value objects, agregados)
- Onion define dónde y cómo organizar esos elementos
- DDD proporciona el lenguaje y conceptos; Onion proporciona la estructura
- Se recomienda usar ambos conjuntamente en sistemas complejos

6. Ventajas Estratégicas

La Arquitectura de Cebolla ofrece beneficios profundos que justifican su adopción en sistemas empresariales complejos.

6.1. Independencia de Frameworks

El núcleo de la aplicación no depende de ningún framework específico. Esto significa:

- Los frameworks pueden actualizarse o reemplazarse sin reescribir la lógica de negocio
- No se está atado a las decisiones de diseño o limitaciones de un framework particular
- El código de negocio no se contamina con anotaciones o herencia de frameworks
- Se puede migrar a tecnologías emergentes manteniendo el núcleo intacto

6.2. Testabilidad Extrema

La arquitectura facilita pruebas a todos los niveles:

- **Pruebas Unitarias del Dominio:** Rápidas, sin dependencias externas, enfocadas en lógica de negocio
- **Pruebas de Integración:** Verifican comunicación entre capas usando mocks o stubs
- **Pruebas de Contrato:** Aseguran que implementaciones cumplen interfaces definidas
- **Desarrollo Guiado por Pruebas (TDD):** Naturalmente compatible con la arquitectura

6.3. Mantenibilidad a Largo Plazo

Los sistemas construidos con esta arquitectura envejecen mejor:

- Las responsabilidades claras facilitan localizar código para modificarlo
- Los cambios están contenidos en capas específicas
- El código expresa claramente la intención del negocio
- Nuevos desarrolladores comprenden el sistema más rápidamente

6.4. Escalabilidad Arquitectónica

La arquitectura soporta crecimiento en múltiples dimensiones:

- **Escalabilidad Técnica:** Diferentes capas pueden desplegarse y escalar independientemente
- **Escalabilidad de Equipo:** Equipos pueden trabajar en diferentes capas sin interferirse
- **Escalabilidad de Complejidad:** El sistema puede crecer en funcionalidad manteniendo organización

6.5. Flexibilidad Tecnológica

Permite decisiones tecnológicas diferidas y reversibles:

- Se puede iniciar con tecnologías simples y migrar a más sofisticadas después
- Diferentes partes del sistema pueden usar tecnologías diferentes
- Se puede experimentar con nuevas tecnologías en la infraestructura sin riesgo

7. Desafíos y Consideraciones

A pesar de sus beneficios, la Arquitectura de Cebolla presenta desafíos que deben considerarse cuidadosamente antes de su adopción.

7.1. Complejidad Inicial y Curva de Aprendizaje

La adopción de esta arquitectura requiere inversión significativa:

- **Configuración Inicial:** Establecer la estructura de proyectos, configurar inyección de dependencias, y definir interfaces requiere más tiempo que arquitecturas simples
- **Conocimiento Requerido:** Los desarrolladores deben comprender principios SOLID, inversión de dependencias, y patrones de diseño
- **Cambio de Mentalidad:** Requiere abandonar enfoques procedurales o centrados en datos para pensar en términos de dominio y responsabilidades
- **Capacitación del Equipo:** El equipo completo debe entender y comprometerse con los principios arquitectónicos

7.2. Overhead de Código

La arquitectura genera más artefactos de código:

- **Múltiples Representaciones:** Entidades, DTOs, ViewModels representan conceptos similares
- **Mappers Abundantes:** Conversiones entre capas requieren código de mapeo explícito
- **Interfaces Proliferadas:** Cada servicio o repositorio requiere su interfaz
- **Más Archivos y Clases:** La estructura del proyecto es más extensa

Sin embargo, este .ºverhead.es en realidad inversión en claridad, testabilidad y mantenibilidad.

7.3. Riesgo de Sobre-ingeniería

Existe tentación de aplicar la arquitectura inapropiadamente:

- **Aplicaciones CRUD Simples:** Para operaciones básicas de crear, leer, actualizar, eliminar, esta arquitectura puede ser excesiva
- **Prototipos y MVPs:** Cuando se necesita velocidad de desarrollo sobre arquitectura
- **Proyectos de Vida Corta:** Si el sistema no evolucionará, la inversión no se recupera
- **Abstracciones Innecesarias:** Crear interfaces para todo "por si acaso"sin necesidad real

7.4. Rendimiento

El mapeo continuo entre capas puede impactar rendimiento:

- Conversiones constantes entre representaciones consumen CPU y memoria
- Múltiples capas de indirección pueden dificultar optimizaciones del compilador
- En sistemas de altísimo rendimiento, el overhead podría ser significativo
- Sin embargo, en la mayoría de aplicaciones empresariales, el impacto es negligible

7.5. Dificultad de Depuración

La indirección arquitectónica puede complicar la depuración:

- Seguir el flujo de ejecución a través de múltiples capas y abstracciones
- Identificar qué implementación concreta se está usando cuando hay múltiples
- Stack traces más largos y complejos
- Requiere herramientas de depuración sofisticadas

8. Criterios de Adopción

Decidir cuándo aplicar la Arquitectura de Cebolla requiere evaluar múltiples factores del proyecto y organización.

8.1. Indicadores Positivos para Adopción

La arquitectura es particularmente apropiada cuando:

- **Lógica de Negocio Compleja:** El sistema tiene reglas de negocio intrincadas que justifican protección arquitectónica
- **Vida Útil Prolongada:** El sistema se mantendrá y evolucionará durante años
- **Múltiples Interfaces:** Se requiere soportar web, móvil, APIs públicas, integraciones B2B simultáneamente
- **Equipos Grandes:** Múltiples equipos trabajarán en diferentes partes del sistema
- **Incertidumbre Tecnológica:** No se está seguro de las tecnologías finales o se prevén cambios
- **Requisitos de Testabilidad:** Pruebas automatizadas extensivas son críticas
- **Evolución Continua:** Se espera agregar funcionalidad constantemente
- **Dominio Rico:** El problema tiene un dominio bien definido con lenguaje especializado

8.2. Indicadores Negativos contra Adopción

La arquitectura probablemente no es apropiada cuando:

- **Aplicaciones CRUD Puras:** Operaciones simples de base de datos sin lógica compleja
- **Prototipos Rápidos:** Necesidad de validar ideas rápidamente
- **Equipos Pequeños:** Un desarrollador o equipo muy reducido
- **Plazos Extremos:** Presión de tiempo crítica donde velocidad supera calidad
- **Proyectos Descartables:** Scripts, herramientas temporales, o sistemas de vida corta
- **Requisitos Extremos de Rendimiento:** Sistemas en tiempo real donde cada microsegundo cuenta
- **Equipo sin Experiencia:** Desarrolladores sin conocimiento de principios SOLID

8.3. Estrategias de Adopción Gradual

No es necesario adoptar toda la arquitectura inmediatamente:

- **Empezar con el Dominio:** Identificar y aislar la lógica de negocio primero
- **Inversión de Dependencias Selectiva:** Aplicar a componentes críticos inicialmente
- **Evolución Incremental:** Refactorizar hacia la arquitectura progresivamente
- **Módulos Piloto:** Implementar completamente en un módulo para aprender
- **Documentación y Capacitación:** Invertir en educación del equipo paralelamente

9. Patrones Complementarios

La Arquitectura de Cebolla se enriquece cuando se combina con otros patrones y prácticas.

9.1. CQRS (Command Query Responsibility Segregation)

CQRS separa operaciones de lectura y escritura:

- **Commands:** Modifican el estado del sistema, pasan por el dominio completo
- **Queries:** Leen datos, pueden bypassear el dominio para optimización
- **Sinergia con Onion:** Commands usan la arquitectura completa; Queries pueden usar proyecciones optimizadas
- **Beneficios:** Optimización independiente de lecturas y escrituras, escalabilidad mejorada

9.2. Event Sourcing

En lugar de almacenar estado actual, se almacenan eventos:

- Los domain events de Onion se convierten en la fuente de verdad
- El estado actual se reconstruye reproduciendo eventos
- Auditoría completa y natural del sistema
- Compatible con arquitecturas distribuidas y microservicios

9.3. Mediator Pattern

Reduce acoplamiento entre componentes:

- Centraliza comunicación entre casos de uso y handlers
- Facilita aspectos transversales (logging, validación, transacciones)
- Implementaciones como MediatR en .NET simplifican la arquitectura
- Cada caso de uso se vuelve un mensaje independiente

9.4. Repository Pattern

Ya integrado en Onion, pero con variantes:

- **Generic Repository:** Operaciones CRUD comunes en repositorio base
- **Specific Repository:** Cada agregado tiene su repositorio específico
- **Unit of Work:** Coordina transacciones entre múltiples repositorios

9.5. Specification Pattern

Encapsula lógica de consulta reutilizable:

- Define criterios de filtrado en el dominio
- Permite composición de especificaciones complejas
- Mantiene el dominio libre de detalles de consulta
- Facilita testing de lógica de filtrado

10. Consideraciones de Implementación

Aspectos prácticos que surgen al implementar la arquitectura.

10.1. Organización de Proyectos

La estructura física del código debe reflejar la arquitectura:

- **Proyectos Separados por Capa:** Domain, Application, Infrastructure, Presentation como proyectos independientes
- **Referencias Controladas:** Infrastructure y Presentation referencian Application; Application referencia Domain; Domain no referencia nada
- **Shared Kernel:** Componentes verdaderamente compartidos en proyecto separado
- **Tests en Paralelo:** Proyectos de prueba espejean la estructura

10.2. Gestión de Transacciones

Las transacciones deben gestionarse cuidadosamente:

- Típicamente se manejan en la capa de aplicación
- El dominio no debe conocer sobre transacciones
- Usar patrones como Unit of Work para coordinar
- Considerar transacciones distribuidas en arquitecturas de microservicios

10.3. Manejo de Errores

Estrategia de excepciones a través de capas:

- **Domain Exceptions:** Errores de negocio (violación de reglas)
- **Application Exceptions:** Errores de coordinación o validación
- **Infrastructure Exceptions:** Errores técnicos (conexión, timeout)
- **Transformación en Fronteras:** Convertir excepciones al cruzar a presentación

10.4. Versionado de APIs

Cuando el sistema expone APIs:

- Los DTOs permiten mantener múltiples versiones de contratos
- El dominio permanece estable mientras cambian las representaciones externas
- Mappers diferentes para cada versión de API
- Deprecación gradual de versiones antiguas

10.5. Persistencia de Agregados

Estrategias para almacenar agregados complejos:

- **ORM Completo:** Mapear todo el agregado con Entity Framework/Hibernate
- **Serialización:** Almacenar agregados como JSON/XML en bases NoSQL
- **Event Sourcing:** Almacenar eventos en lugar de estado
- **Híbrido:** Combinar enfoques según necesidades

11. Arquitectura de Cebolla en Microservicios

La arquitectura se adapta naturalmente a arquitecturas distribuidas.

11.1. Microservicio como Cebolla

Cada microservicio puede implementar internamente la arquitectura:

- El dominio de cada microservicio está completamente aislado
- La comunicación entre microservicios ocurre en la capa de infraestructura
- Cada servicio mantiene su propia base de datos (database per service)
- Los bounded contexts de DDD mapean a microservicios

11.2. Comunicación entre Servicios

Estrategias para integración:

- **REST APIs:** Comunicación síncrona mediante HTTP
- **Message Queues:** Comunicación asíncrona mediante eventos
- **Event-Driven:** Domain events publican a bus de mensajes
- **API Gateway:** Fachada que orquesta múltiples servicios

11.3. Consistencia Eventual

En sistemas distribuidos, la consistencia inmediata es imposible:

- Los agregados dentro de un servicio mantienen consistencia fuerte
- Entre servicios se acepta consistencia eventual
- Sagas coordinan transacciones distribuidas
- El dominio debe diseñarse tolerando estados temporalmente inconsistentes

12. Conclusiones

La Arquitectura de Cebolla representa un enfoque maduro y profundamente fundamentado para el diseño de aplicaciones empresariales complejas. Su énfasis riguroso en la inversión de dependencias, la separación estricta de responsabilidades, y la protección del dominio de negocio resulta en sistemas que pueden evolucionar, mantenerse y adaptarse a lo largo del tiempo de manera sostenible.

A través de este estudio exhaustivo, se ha demostrado cómo esta arquitectura trasciende ser simplemente un patrón de organización de código para convertirse en una filosofía de diseño que prioriza la lógica de negocio sobre los detalles técnicos. Al colocar el dominio en el centro y hacer que todas las dependencias apunten hacia él, se crea un sistema donde los cambios tecnológicos no amenazan la estabilidad del núcleo empresarial.

Los fundamentos teóricos en principios SOLID, especialmente el Principio de Inversión de Dependencias, proporcionan una base sólida que ha resistido la prueba del tiempo. La regla fundamental de que las dependencias solo pueden apuntar hacia adentro crea una jerarquía arquitectónica que naturalmente previene el acoplamiento indeseable y promueve el código limpio y testeable.

La estructura en capas concéntricas no es meramente metafórica; representa una separación real y tangible de responsabilidades donde cada capa tiene un propósito claro y bien definido. El dominio puro en el centro, libre de cualquier contaminación tecnológica, puede expresar las reglas de negocio en el lenguaje del dominio mismo, facilitando la colaboración con expertos del negocio y asegurando que el código refleje fielmente las necesidades empresariales.

Sin embargo, es crucial reconocer que la Arquitectura de Cebolla no es una solución universal aplicable indiscriminadamente a todos los proyectos. Su adopción implica costos en términos de complejidad inicial, curva de aprendizaje del equipo, y overhead de código. Estos costos son inversiones que se recuperan con el tiempo en sistemas de vida larga con requisitos evolutivos, pero pueden ser prohibitivos en proyectos pequeños, prototipos, o aplicaciones CRUD simples donde la lógica de negocio es trivial.

La decisión de adoptar esta arquitectura debe basarse en un análisis cuidadoso de factores como la complejidad del dominio, la vida útil esperada del sistema, el tamaño y experiencia del equipo, los requisitos de testabilidad, y la probabilidad de cambios tecnológicos futuros. Cuando estos factores se alinean favorablemente, la Arquitectura de Cebolla proporciona un retorno de inversión significativo en términos de mantenibilidad, testabilidad, y flexibilidad.

La comparación con otras arquitecturas revela que Onion no existe en aislamiento sino como parte de un ecosistema de ideas relacionadas. Su estrecha relación con la Arquitectura Hexagonal, Clean Architecture, y Domain-Driven Design muestra cómo diferentes pensadores han convergido en principios similares desde perspectivas ligeramente diferentes. Esta convergencia fortalece la confianza en que estos principios representan genuinamente mejores prácticas para sistemas complejos.

Los patrones complementarios como CQRS, Event Sourcing, y el patrón Mediator no son requisitos sino herramientas adicionales que pueden potenciar la arquitectura cuando las necesidades del sistema lo justifican. La belleza de la Arquitectura de Cebolla es que proporciona una base sólida sobre la cual estos patrones pueden incorporarse naturalmente sin violación de principios fundamentales.

En el contexto actual de microservicios y sistemas distribuidos, la arquitectura demuestra su relevancia continua. Cada microservicio puede implementar internamente Onion Architecture, manteniendo su dominio protegido mientras participa en un ecosistema distribuido más amplio. Esta escalabilidad arquitectónica desde el monolito hasta sistemas distribuidos complejos subraya la versatilidad del enfoque.

Para equipos y organizaciones que decidan adoptar esta arquitectura, se recomienda una aproximación gradual y pragmática. Comenzar con módulos o componentes críticos donde el beneficio sea más claro, invertir en capacitación del equipo, y permitir que la comprensión y adopción evolucionen orgánicamente. El compromiso debe ser con los principios más que con reglas rígidas, adaptando la implementación a las realidades específicas del proyecto y organización.

En conclusión, la Arquitectura de Cebolla es una herramienta poderosa y probada en el arsenal del arquitecto de software moderno. No es la única herramienta, ni siempre la apropiada, pero cuando se aplica juiciosamente a sistemas empresariales complejos donde la mantenibilidad a largo plazo, la testabilidad exhaustiva, y la flexibilidad tecnológica son prioritarias, proporciona un marco arquitectónico que permite construir sistemas que no solo funcionan hoy, sino que pueden evolucionar exitosamente durante años o décadas. Su adopción requiere disciplina, comprensión profunda de principios de diseño, y compromiso del equipo, pero los beneficios en términos de calidad, sostenibilidad, y adaptabilidad del código justifican plenamente la inversión inicial para los sistemas apropiados.

Segura Gutierrez Obed

Capa	Responsabilidad	Dependencias
Dominio (Core)	Entidades, Value Objects, Reglas de negocio, Interfaces de repositorio	Ninguna (cero dependencias externas)
Aplicación	Casos de uso, DTOs, Orquestación, Coordinación de transacciones	Solo Dominio
Infraestructura	Implementación de repositorios, Acceso a BD, Servicios externos, Logging	Dominio, Aplicación
Presentación	Controladores, ViewModels, API Endpoints, Validación de entrada	Aplicación

Cuadro 1: Resumen de Capas en Arquitectura de Cebolla

13. Referencias Bibliográficas

Referencias

[Martin, 2017] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design* Prentice Hall, 2017.

[Palermo, 2008] Jeffrey Palermo. *The Onion Architecture: Part 1, Part 2, Part 3, Part 4* Blog Posts, 2008-2013. Disponible en: https://jeffreypalermo.com/

[Fowler, 2002] Martin Fowler. *Patterns of Enterprise Application Architecture* Addison-Wesley Professional, 2002.

[Vernon, 2013] Vaughn Vernon. *Implementing Domain-Driven Design* Addison-Wesley Professional, 2013.

[Evans, 2003] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software* Addison-Wesley Professional, 2003.

[Cockburn, 2005] Alistair Cockburn. *Hexagonal Architecture* Article, 2005. Disponible en: https://alistair.cockburn.us/

[Nilsson, 2006] Jimmy Nilsson. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET* Addison-Wesley Professional, 2006.

[Freeman, 2009] Steve Freeman, Nat Pryce. *Growing Object-Oriented Software, Guided by Tests* Addison-Wesley Professional, 2009.

[Gamma, 1994] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional, 1994.

[Microsoft, 2024] Microsoft. *Common web application architectures - Clean Architecture* Microsoft Docs, 2024. Disponible en: https://docs.microsoft.com/

[Seemann, 2019] Mark Seemann. *Dependency Injection Principles, Practices, and Patterns* Manning Publications, 2019.

[Richardson, 2018] Chris Richardson. *Microservices Patterns: With examples in Java* Manning Publications, 2018.