Criterio topológico para la elección del modelo de persistencia

Fase de diseño

RUP Simplificado

# Resumen Ejecutivo (Abstract)

En este trabajo se propone un criterio concreto para la elección del modelo de persistencia en aplicaciones de software, basado en la topología lógica de las relaciones entre entidades del dominio. Se parte de una hipótesis clara: cuando dicha topología forma un árbol, conviene utilizar un modelo documental; cuando forma un grafo, es preferible un modelo relacional. Esta hipótesis se valida con un caso práctico en lenguaje C#, implementado con dos tecnologías ampliamente difundidas: SQLite (relacional) y LiteDB (documental).

El estudio se enfoca en cómo distintas representaciones afectan el diseño y la complejidad del código de acceso a datos, y los resultados empíricos muestran una reducción del 66 % en la cantidad de código necesario cuando se elige el modelo adecuado según la topología. Este criterio no depende de un lenguaje ni tecnología específica, y puede aplicarse a otros entornos como PostgreSQL, MongoDB o bases orientadas a grafos como Neo4j.

Además de validar su utilidad, este trabajo busca restituir al diseño estructural un rol central en la arquitectura de software, contribuyendo con una herramienta analítica que permite tomar decisiones informadas en etapas tempranas del desarrollo.

# **Contribuciones del trabajo**

1. **Propuesta de un criterio topológico para la elección del modelo de persistencia.**  
   Se plantea que la estructura lógica de las relaciones entre entidades (árbol vs. grafo) debe guiar la selección entre modelos documentales y relacionales.
2. **Validación empírica del criterio mediante un caso práctico real.**  
   Se demuestra la aplicabilidad del criterio en un sistema con lógica jerárquica (roles, permisos y excepciones), comparando en detalle implementaciones relacional y documental.
3. **Integración del criterio en el ciclo de desarrollo orientado a objetos.**  
   Se muestra cómo el criterio se alinea con las fases clásicas de análisis, diseño e implementación, sin necesidad de introducir modelos nuevos o herramientas externas.
4. **Justificación de arquitecturas híbridas estructuradas.**  
   Se establece que distintos módulos pueden usar distintas tecnologías de persistencia según su topología, sin sacrificar coherencia ni mantenibilidad.
5. **Respuesta técnica a objeciones comunes.**  
   Se abordan críticas frecuentes con argumentos fundados, mostrando la flexibilidad del criterio sin reducirlo a una elección dogmática.
6. **Coherencia con principios de diseño (SOLID).**  
   Se verifica que el criterio promueve responsabilidades claras, bajo acoplamiento, modularidad y estabilidad semántica en el modelo de dominio.
7. **Desmitificación del predominio relacional.**  
   Se desmontan mitos habituales sobre la superioridad del modelo relacional, sin descalificarlo, sino posicionándolo como solución adecuada para dominios con topología de grafo.

# 1. Introducción

## 1.1. El problema de las ineficiencias por la tradición relacional

La preferencia histórica por las bases de datos relacionales en sistemas orientados a objetos genera ineficiencias que afectan el rendimiento, la escalabilidad y la mantenibilidad. Esta tradición, basada en sistemas de gestión de bases de datos relacionales (RDBMS), choca con las estructuras naturales de lenguajes como C# o Java, añadiendo complejidad innecesaria.

Un problema clave es la necesidad de JOINs complejos para reconstruir relaciones que en un modelo de objetos serían referencias directas. Por ejemplo, en un sistema con clientes y facturas, una base relacional usa múltiples tablas (Clientes, Facturas, DetallesFactura) y consultas con JOINs, como:

SELECT c.Id, c.Nombre, f.IdFactura, f.Fecha, i.IdItem, i.Descripcion

FROM Clientes c

LEFT JOIN Facturas f ON c.Id = f.IdCliente

LEFT JOIN ItemsFactura i ON f.IdFactura = i.IdFactura

WHERE c.Id = @ClienteId;

Estas consultas aumentan la latencia y complican el mantenimiento, especialmente con grandes volúmenes de datos o estructuras jerárquicas profundas.

Otro inconveniente es el mapeo objeto-relacional (ORM). Herramientas como Entity Framework reducen la brecha entre objetos y tablas, pero introducen sobrecarga y obligan a los desarrolladores a lidiar con conflictos (como herencia) y consultas sub-óptimas, desviando esfuerzo de la lógica de negocio a la persistencia.

En contraste, modelos no relacionales como bases documentales (ej. MongoDB) representan estas estructuras de forma más natural. Un cliente con facturas podría ser un solo documento JSON:

{

"Id": "C001",

"Nombre": "Cliente Ejemplo",

"Facturas": [

{

"IdFactura": "F001",

"Fecha": "2025-04-20",

"Items": [

{ "IdItem": "I001", "Descripcion": "Producto A" },

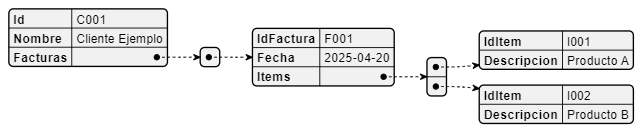
{ "IdItem": "I002", "Descripcion": "Producto B" }

]

}

]

}

[[1]](#footnote-1)

Esto elimina JOINs, simplifica consultas y alinea la persistencia con el modelo de objetos. Así, la elección automática de bases relacionales, sin evaluar la topología de los datos, perpetúa ineficiencias evitables con enfoques más acordes al paradigma orientado a objetos.

## 1.2. La hipótesis del criterio basado en topología

Este trabajo propone un criterio basado en topología para seleccionar el modelo de persistencia más adecuado en el diseño orientado a objetos, utilizando la estructura de las relaciones entre entidades como guía principal. Específicamente, el criterio analiza si las relaciones del dominio forman una **estructura de árbol** (jerárquica, sin ciclos) o un **grafo** (con ciclos o relaciones muchos-a-muchos). Con base en esta evaluación, se sugiere que:

* Las estructuras de árbol se alinean mejor con **modelos documentales** (por ejemplo, MongoDB), que manejan eficientemente datos jerárquicos y anidados.
* Los grafos, con sus interconexiones complejas, se benefician de **modelos relacionales** (RDBMS) o **bases de grafos** (como Neo4j), según la naturaleza de las relaciones.

**Ejemplo práctico:** Consideremos un sistema de gestión de clientes donde cada cliente tiene múltiples facturas, y cada factura incluye ítems. Esta estructura arbórea se representa naturalmente en un modelo documental, eliminando la necesidad de JOINs complejos y reflejando las colecciones anidadas típicas de lenguajes como C#. Por otro lado, si los clientes compartieran facturas entre sí (relaciones cruzadas), un modelo relacional sería más adecuado para gestionar esas interconexiones.

La hipótesis central afirma que un análisis topológico del dominio permite alinear el diseño de las clases con el modelo de persistencia, optimizando el rendimiento y reduciendo ineficiencias. Este enfoque desafía la preferencia histórica por bases de datos relacionales como solución universal y propone una alternativa más natural para el paradigma orientado a objetos. Al identificar si las entidades forman árboles o grafos, se pueden tomar decisiones informadas que minimicen la sobrecarga computacional, simplifiquen las operaciones de lectura/escritura y mejoren la mantenibilidad del sistema.

En resumen, este criterio no solo mejora la eficiencia del software, sino que también asegura que la estructura de los objetos en memoria sea compatible con el modelo de persistencia elegido. Así, la propuesta representa un avance hacia diseños más robustos, escalables y alineados con los principios del diseño orientado a objetos.

## 1.3. Relevancia y público objetivo

El criterio basado en topología propuesto en este artículo es de particular relevancia para **arquitectos de software y desarrolladores** que trabajan en proyectos orientados a objetos utilizando lenguajes como C#, y que emplean herramientas y metodologías como el **Lenguaje Unificado de Modelado (UML)** y procesos de desarrollo basados en **RUP (Rational Unified Process) simplificada**. Estos profesionales enfrentan el desafío de diseñar sistemas que sean eficientes, escalables y mantenibles, y el criterio ofrece una guía estructurada para seleccionar el modelo de persistencia más adecuado según la topología de las relaciones del dominio, optimizando así el diseño desde las primeras etapas.

La relevancia del criterio se manifiesta tanto en su **valor académico** como en su **aplicación práctica**. Desde el punto de vista académico, la propuesta introduce una novedad al priorizar un enfoque basado en la topología de los datos, desafiando la hegemonía de los modelos relacionales y promoviendo una reevaluación de los paradigmas de persistencia en el diseño orientado a objetos. Este enfoque fomenta nuevas líneas de investigación sobre cómo las estructuras de datos influyen en la eficiencia de los sistemas. Prácticamente, el criterio permite a los equipos de desarrollo **evitar ineficiencias** asociadas con elecciones de persistencia sub-óptimas, como JOINs innecesarios o mapeos objeto-relacionales complejos. Además, al proporcionar un marco para evaluar y justificar la elección del modelo de persistencia, facilita la **transición entre modelos** (por ejemplo, de relacional a NoSQL) en etapas posteriores del proyecto, reduciendo costos y riesgos asociados con refactorizaciones.

El alcance del criterio se delimita específicamente a la **etapa de diseño** del desarrollo de software. No se centra en la implementación práctica (programación) ni en la selección de tecnologías específicas (por ejemplo, frameworks o bases de datos particulares), sino en la definición de un modelo conceptual que alinee la estructura del dominio con el modelo de persistencia. Esta focalización asegura que el criterio sea aplicable en una amplia variedad de contextos, independientemente de las herramientas tecnológicas utilizadas, y que proporcione una base sólida para tomar decisiones informadas en las fases iniciales del ciclo de vida del software.

# 2. Contexto: Diseño Orientado a Objetos en RUP Simplificado

## 2.1. El proceso de RUP simplificado

El proceso unificado racional (RUP), en su versión simplificada, estructura el desarrollo en etapas iterativas con actividades recurrentes: análisis, diseño, implementación y pruebas. Estas fases no son secuenciales, sino que se superponen e iteran según necesidades del proyecto. En este trabajo se adopta esta versión simplificada, manteniendo un enfoque disciplinado que permite trazar el recorrido completo desde el análisis orientado a objetos hasta la programación efectiva.

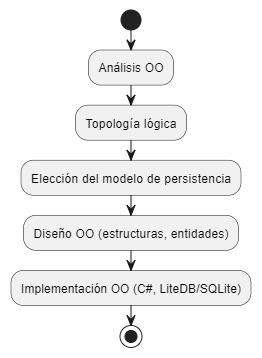


Figure 1 - Flujo simplificado del proceso unificado aplicado en este trabajo.[[2]](#footnote-2)

Como se ve en la imagen, el trabajo sigue un proceso iterativo donde el análisis estructural (topológico) tiene un rol protagónico en la transición desde el diseño hacia la implementación.

## 2.2. El rol de la arquitectura

Transversal a todas las fases, la arquitectura tiene la responsabilidad de **garantizar la adaptabilidad** del sistema frente a **cambios imprevisibles**. Esto implica definir patrones y principios (por ejemplo, modularidad, desacoplamiento) que permitan ajustar el modelo de persistencia o las clases ante requisitos no anticipados, como la integración de nuevos tipos de datos o la migración a otro modelo de base de datos. La arquitectura asegura que las decisiones de diseño y programación no limiten la evolución futura del sistema.

La clara delimitación de estas responsabilidades asegura que el criterio basado en topología se integre de manera efectiva en el proceso, optimizando la elección del modelo de persistencia en la fase de diseño mientras se mantiene la flexibilidad para adaptarse a cambios en las fases subsiguientes.

## 2.3. El rol del diseño

El diseño orientado a objetos desempeña un papel central en el desarrollo de software bajo el marco de la RUP simplificada, sirviendo como el puente entre los requerimientos identificados en el análisis y la implementación técnica en la programación. Este proceso utiliza el **Lenguaje Unificado de Modelado (UML)** para representar de manera precisa y visual las **clases** y sus **relaciones**, proporcionando una base estructurada para modelar el dominio del problema. Los diagramas UML, como los de clases, objetos y secuencias, permiten a los arquitectos y desarrolladores definir atributos, métodos y asociaciones (por ejemplo, herencia, composición, agregación) que reflejan las entidades y dinámicas del sistema. Por ejemplo, en un sistema de gestión de clientes, un diagrama de clases podría modelar la relación jerárquica entre Cliente y Factura, especificando colecciones anidadas o referencias entre objetos.

Un aspecto fundamental del diseño orientado a objetos es su compromiso con los **principios SOLID** (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion). Estos principios guían la creación de clases cohesivas, modulares y extensibles, asegurando que el modelo del dominio sea robusto y mantenible antes de considerar aspectos técnicos como la persistencia.

Al priorizar el modelado del dominio, el diseño garantiza que las clases representen fielmente las entidades y relaciones del mundo real, evitando distorsiones introducidas por restricciones de un modelo específico de persistencia. Por ejemplo, una clase Cliente se diseña para encapsular su comportamiento y datos (como nombre y facturas asociadas) sin asumir inicialmente si se persistirá en una base relacional o NoSQL.

En el contexto del criterio basado en topología propuesto, el diseño orientado a objetos asume una responsabilidad adicional: **elegir el modelo de persistencia** para alinearse con la topología de las relaciones. Una vez identificada la topología (por ejemplo, árbol para estructuras jerárquicas o grafo para relaciones complejas), el diseño evalúa si un modelo jerárquico/NoSQL (como MongoDB) o relacional/grafos (como SQL Server o Neo4j) es más adecuado.

Posteriormente, se ajustan las clases para optimizar la persistencia según el modelo inferido, por ejemplo, estructurando colecciones anidadas para bases documentales o definiendo claves foráneas para bases relacionales. Este proceso asegura que la estructura de los objetos en memoria sea compatible con el modelo de persistencia elegido, minimizando la sobrecarga de mapeo y mejorando el rendimiento de las operaciones de lectura y escritura.

Así, el diseño orientado a objetos no solo modela el dominio con precisión y adherencia a principios sólidos, sino que también actúa como el punto de inflexión donde se toman decisiones estratégicas sobre la persistencia, alineando la arquitectura del sistema con las necesidades del dominio y los objetivos de eficiencia.

# 3. La Hipótesis: Criterio Basado en Topología

## 3.1. El flujo de decisión del criterio

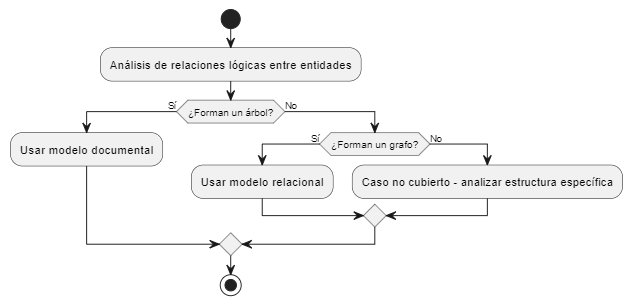
El criterio basado en topología propone un enfoque sistemático para seleccionar el modelo de persistencia en el diseño orientado a objetos, basándose en la estructura de las relaciones entre las entidades del dominio. Este criterio clasifica las topologías de datos en dos categorías principales: **árboles** (estructuras jerárquicas sin ciclos) y **grafos** (estructuras con ciclos o relaciones muchos-a-muchos). La elección del modelo de persistencia se guía por un flujo de decisión claro que prioriza la eficiencia y la alineación con el paradigma orientado a objetos.

**Explicación del criterio**

* **Árbol (sin ciclos)**: Si las relaciones entre entidades forman una estructura arbórea, donde cada entidad tiene un único "padre" y no existen caminos cerrados, se selecciona un modelo **jerárquico/NoSQL**, como una base de datos documental (por ejemplo, MongoDB). Estas bases son ideales para representar jerarquías, ya que permiten almacenar datos anidados que reflejan directamente las colecciones de objetos en lenguajes como C#, eliminando la necesidad de JOINs y simplificando las consultas. Por ejemplo, un árbol de clientes y facturas se mapearía naturalmente a documentos JSON anidados.
* **Grafo (ciclos, muchos-a-muchos)**: Si las relaciones incluyen **ciclos** (caminos cerrados entre entidades) o relaciones **muchos-a-muchos**, se opta por un modelo **relacional** (por ejemplo, SQL Server) o de **grafos** (por ejemplo, Neo4j). Estos modelos son más adecuados para manejar redes complejas donde las entidades están interconectadas de manera no jerárquica. Por ejemplo, un sistema donde los clientes comparten facturas o forman redes de referencias requeriría un modelo que gestione eficientemente estas relaciones cruzadas.

Un **ciclo** se define como un camino cerrado en las relaciones entre entidades, como en el caso de Cliente A → Factura 1 → Cliente B → Factura 2 → Cliente A. Este tipo de estructura indica una topología de grafo que no puede representarse eficientemente en un modelo jerárquico sin redundancias o complejidad adicional. El flujo de decisión del criterio se representa en el siguiente diagrama, que guía la selección del modelo de persistencia mediante preguntas secuenciales:

1. **Pregunta 1: ¿Es un árbol?** Si las relaciones forman una estructura jerárquica sin ciclos (por ejemplo, un cliente con facturas anidadas), se elige un modelo jerárquico/NoSQL. Si no es un árbol, se pasa a la siguiente pregunta.
2. **Pregunta 2: ¿Hay ciclos o relaciones muchos-a-muchos?** Si existen ciclos o relaciones muchos-a-muchos (por ejemplo, clientes compartiendo facturas), se selecciona un modelo relacional o de grafos. Si no hay ciclos ni relaciones complejas, se reevalúa el diseño.

[[3]](#footnote-3)

En dominios donde la topología lógica adopta una forma claramente reticulada o con ciclos, el modelo relacional suele ser adecuado. No obstante, en casos extremos de grafos densos, podrían considerarse alternativas como Neo4j o JanusGraph, aunque no fueron contempladas en este trabajo.

**Conclusión**

Este flujo de decisión asegura que la elección del modelo de persistencia sea coherente con la topología de los datos, optimizando el diseño para reflejar las relaciones del dominio y minimizando ineficiencias en la persistencia. En la siguiente sección, se explorarán ejemplos prácticos para ilustrar la aplicación de este criterio en sistemas reales.

## 3.2. Preferencia por modelos no relacionales

La preferencia por **modelos no relacionales**, particularmente los jerárquicos como las bases de datos documentales (por ejemplo, MongoDB, LiteDB), se fundamenta en su capacidad para representar de manera natural las estructuras arbóreas identificadas en el diseño orientado a objetos. Estas estructuras, caracterizadas por relaciones jerárquicas sin ciclos, son comunes en muchos dominios, como sistemas de clientes y facturas o árboles de categorías. Los modelos no relacionales ofrecen ventajas significativas en términos de **reducción de complejidad** y **minimización del código**, alineándose estrechamente con las estructuras de datos utilizadas en lenguajes orientados a objetos.

Los modelos jerárquicos son más naturales para árboles porque permiten almacenar datos en una estructura anidada que refleja directamente las relaciones padre-hijo del dominio. Por ejemplo, en un sistema donde un cliente tiene múltiples facturas, y cada factura contiene ítems, un modelo documental puede representar esta jerarquía como un único documento JSON con listas anidadas. Esto elimina la necesidad de **JOINs** y reduce la complejidad del mapeo objeto-relacional, ya que los objetos en memoria (por ejemplo, una clase Cliente con una lista de Facturas) se serializan y deserializan directamente sin transformaciones adicionales. En contraste, un modelo relacional requeriría múltiples tablas y consultas complejas, aumentando tanto el esfuerzo de desarrollo como la latencia en tiempo de ejecución.

Un experimento comparativo ilustra esta ventaja de manera concreta. En un caso de estudio, se implementó un sistema jerárquico simple (un árbol de clientes y pedidos) utilizando **SQLite** (base relacional) y **LiteDB** (base documental). La implementación en SQLite requirió aproximadamente **454 líneas de código** que incluyen: **mappers**, **records**, **context** y **repository**. Por otro lado, la implementación en **LiteDB** necesitó **151 líneas de código** gracias a la capacidad de almacenar documentos JSON que reflejaban directamente las clases de C#, por lo cual solo requirió: **context** y **repository**. Esta diferencia no solo destaca la reducción en la complejidad del código, sino también un menor tiempo de desarrollo y una mayor facilidad de mantenimiento.

Además, los modelos no relacionales se alinean de manera natural con las estructuras de datos en lenguajes como C#. En este lenguaje, las relaciones jerárquicas se modelan frecuentemente mediante **listas anidadas** o colecciones (por ejemplo, List<Factura> dentro de una clase Cliente). Los modelos documentales permiten persistir estas colecciones sin necesidad de descomponerlas en tablas separadas, preservando la integridad del diseño orientado a objetos. Esta correspondencia directa entre el modelo en memoria y el modelo de persistencia reduce la impedancia entre el código y la base de datos, facilitando operaciones como la lectura, escritura y actualización de datos.

En resumen, la preferencia por modelos no relacionales en estructuras arbóreas se justifica por su simplicidad, eficiencia y alineación con los paradigmas de programación orientada a objetos. La evidencia empírica y la compatibilidad con estructuras de C# refuerzan la hipótesis de que estos modelos son la elección óptima cuando la topología de los datos es jerárquica, desafiando la tradición relacional y promoviendo un diseño más eficiente y natural.

## 3.3. Adecuación de clases para persistencia

La **adecuación de clases** en el diseño orientado a objetos consiste en ajustar las clases modeladas en UML para alinearlas con el modelo de persistencia seleccionado, guiados por los principios de **eficiencia** y **claridad del dominio**. Este proceso asegura que la estructura de las clases sea compatible con el modelo de persistencia (jerárquico/NoSQL o relacional/grafos), minimizando la complejidad de las operaciones de persistencia y preservando la integridad del diseño orientado a objetos. La adecuación se realiza en la fase de diseño, tras identificar la topología de las relaciones, y respeta los principios **SOLID** para mantener un modelo robusto y mantenible.

**Ejemplo jerárquico**

En un modelo jerárquico, como un sistema que persiste datos en XML o una base documental, las clases deben evitar referencias circulares que compliquen la serialización. Por ejemplo, consideremos un dominio con las entidades Persona y Auto, donde cada auto tiene un dueño (una persona). Una representación inicial podría incluir una referencia directa en Auto a un objeto Persona (por ejemplo, Persona Dueño). Sin embargo, esto puede generar ciclos si Persona también referencia una lista de Auto. Para adecuar la clase, se reemplaza Persona Dueño por un identificador simple, int DueñoId, que mantiene la relación sin introducir dependencias circulares. El código en C# sería:

public class Auto

{

public int Id { get; set; }

public string Modelo { get; set; }

// public Persona Dueño { get; set; } <- Reemplazamos.

public int DueñoId { get; set; }

}

Este ajuste permite serializar Auto en XML o JSON sin problemas, manteniendo la relación con Persona a través del identificador y respetando la claridad del dominio.

**Ejemplo relacional (escala pequeña)**

En un modelo relacional para escalas pequeñas, donde la eficiencia y simplicidad son prioritarias, las clases pueden ajustarse para evitar estructuras complejas. Por ejemplo, en un sistema donde una Persona puede tener varios Auto, una representación inicial podría incluir una List<Auto> en Persona. Sin embargo, para una escala pequeña (por ejemplo, hasta 10 autos por persona) con una relación optativa, es más eficiente almacenar un arreglo de identificadores, string[] AutosId, en lugar de una tabla intermedia. Esto reduce la complejidad del esquema relacional y agiliza las consultas. El código en C# sería:

public class Persona

{

public int Id { get; set; }

public string Nombre { get; set; }

// public List<Auto> Autos { get; set; } <- Reemplazamos.

public string[] AutosId { get; set; }

}

Este diseño es adecuado para escenarios donde las relaciones son simples y el volumen de datos es limitado, **preservando la claridad del dominio** sin introducir sobrecarga innecesaria.

**Ejemplo relacional (escala grande)**

En escenarios de gran escala (aquí sí abordamos el modelo relacional tradicional), como una Empresa con 500 autos, donde la relación entre Empresa y Auto es obligatoria, la adecuación requiere un enfoque diferente. En lugar de almacenar una lista de identificadores en Empresa, es más eficiente usar una **clave foránea** (EmpresaId) en la clase Auto para reflejar la relación en una base relacional. Esto optimiza las consultas y asegura la integridad referencial, especialmente en sistemas con grandes volúmenes de datos. La clase Auto podría definirse como:

public class Auto

{

public int Id { get; set; }

public string Modelo { get; set; }

public int EmpresaId { get; set; }

}

Este diseño refleja la necesidad de escalabilidad y asegura que las operaciones de persistencia sean eficientes en bases relacionales.

**Principios de la adecuación**

La adecuación de clases respeta los principios **SOLID**, asegurando que las clases mantengan una **única responsabilidad**, sean **abiertas a la extensión pero cerradas a la modificación**, y utilicen **interfaces segregadas** cuando sea necesario. Además, prioriza el **dominio** al preservar la semántica de las entidades y relaciones, ajustando solo los aspectos técnicos necesarios para la persistencia. Por ejemplo, reemplazar una lista de objetos por un arreglo de identificadores no altera la lógica de negocio, pero optimiza la interacción con el modelo de persistencia.

En conclusión, la adecuación de clases es un paso crítico para alinear el diseño orientado a objetos con el modelo de persistencia, garantizando eficiencia y claridad. Los ejemplos jerárquico y relacional demuestran cómo los ajustes específicos a la topología y escala del sistema pueden mejorar el rendimiento sin comprometer la calidad del diseño.

## 3.4. Límites del criterio basado en topología

El criterio basado en topología es una herramienta útil para decidir qué modelo de persistencia usar (por ejemplo, relacional o documental) según la estructura de los datos. Sin embargo, no es una solución mágica que funcione en todos los casos. Hay situaciones donde otros factores, como la integridad referencial, la escala del sistema, los requisitos de análisis avanzado o las necesidades transaccionales, pueden hacer que este criterio no sea suficiente por sí solo.

**Integridad referencial estricta**

En algunos sistemas, es fundamental garantizar que las relaciones entre datos sean consistentes y obligatorias. Imaginemos un sistema de facturación: cada factura debe estar vinculada a un cliente que realmente exista. Aunque la estructura parezca un árbol simple (Cliente → Factura), una base de datos relacional con claves foráneas es ideal aquí porque asegura esa integridad de forma nativa. En cambio, un modelo documental (como MongoDB) no ofrece esas garantías integradas, lo que podría generar problemas si, por error, una factura queda "huérfana" sin cliente asociado.

**Escala y rendimiento**

Cuando se trabaja con estructuras jerárquicas muy grandes, como un árbol de categorías con miles de nodos (por ejemplo, un catálogo de una tienda online), el rendimiento puede ser un factor decisivo. El reorganizar la jerarquía frecuentemente o hacer búsquedas profundas (por ejemplo, encontrar todos los nodos a cierto nivel), una base relacional con índices bien diseñados suele ser más eficiente. En un modelo documental, estas operaciones podrían requerir desnormalizar los datos o escribir consultas más complejas, lo que afecta el rendimiento y complica el mantenimiento.

**Análisis cruzado y reportes**

Pongamos un ejemplo concreto: un sistema de gestión de inventarios con una estructura arbórea (Categoría → Producto → Lote). Si el negocio necesita reportes avanzados, como calcular el total de productos por categoría y por proveedor, un modelo relacional brilla por su capacidad para hacer consultas agregadas de forma eficiente usando JOINs. En un modelo documental, lograr lo mismo podría implicar varias consultas separadas o incluso depender de herramientas externas para procesar los datos, lo que añade complejidad y tiempo.

**Transacciones entre módulos**

En sistemas más complejos, donde diferentes partes usan modelos distintos (por ejemplo, un módulo de facturación con base documental y otro de cobranzas con base relacional), las transacciones cruzadas pueden ser un dolor de cabeza. Actualizar una factura y registrar un pago al mismo tiempo: garantizar que ambas operaciones se completen (o fallen juntas) es complicado si no hay soporte nativo para transacciones distribuidas. Esto muestra que el criterio topológico no siempre considera las necesidades de integración entre componentes del sistema.

**Conclusión**

El criterio basado en topología es un buen punto de partida para elegir un modelo de persistencia, pero no lo es todo. Debes complementarlo con un análisis detallado de los requisitos específicos de tu sistema: ¿qué tan crítica es la integridad? ¿Qué escala manejas? ¿Qué tipo de análisis necesitas? La arquitectura debe ser flexible para adaptarse a estas situaciones, permitiendo ajustes o incluso migraciones parciales si las circunstancias cambian.

# 4. Ejemplos Ilustrativos

## 4.1. Caso 1: Ventajas del modelo arbóreo

Este caso analiza un sistema de facturación para una empresa de mudanzas. Cada cliente tiene varias facturas, y cada factura está asociada a un único ítem (servicio de mudanza). La estructura es un árbol puro: Cliente → Factura → Ítem, sin ciclos ni relaciones cruzadas. Un modelo documental (como XML o MongoDB) resulta ideal porque refleja esta jerarquía de forma natural.

**Ventajas:**

* **Simplicidad:** Toda la información de un cliente (facturas e ítems) se almacena en un solo documento, evitando operaciones complejas como JOINs.
* **Eficiencia:** Consultar todos los datos de un cliente es rápido y directo.
* **Alineación con programación orientada a objetos:** La estructura anidada se mapea fácilmente a colecciones en lenguajes como C#.

**Desventajas potenciales:**

* **Escalabilidad limitada:** Si un cliente acumula miles de facturas, el documento puede crecer demasiado y afectar el rendimiento.
* **Actualizaciones parciales:** Modificar un ítem específico dentro de una factura anidada puede ser menos eficiente que en un modelo relacional.

**Conclusión:** Para jerarquías simples y estables, el modelo documental es una opción sólida, pero debe evaluarse el crecimiento futuro.

Otro aspecto favorable del modelo documental en este contexto es su eficiencia al recuperar subárboles completos, como las facturas de un cliente o los ítems de una orden. Esta alineación entre la estructura lógica y la representación física permite lecturas naturales sin joins ni transformaciones adicionales.

## 4.2. Caso 2: Límites del modelo arbóreo

Aquí examinamos un sistema de gestión de autos, donde cada persona puede poseer varios vehículos, formando un árbol: Persona → Auto. Aunque la estructura sugiere un modelo documental, los requisitos de integridad referencial (asegurar que cada auto tenga un propietario válido) y la necesidad de actualizaciones consistentes favorecieron un modelo relacional.

**Razones para elegir un modelo relacional:**

* **Integridad:** Las claves foráneas garantizan que cada auto esté vinculado a una persona existente.
* **Transacciones:** Operaciones como transferir la propiedad de un auto se manejan de forma segura con transacciones ACID.

**Comparación con el modelo documental:**

* En un modelo documental, la integridad referencial no es nativa y depende del código, lo que podría causar inconsistencias.
* Para relaciones simples y de baja cardinalidad (pocas personas con pocos autos), un arreglo de identificadores (como string[] AutosId) en la entidad Persona podría funcionar bien y ser más simple.

**Conclusión:** Aunque la topología apunte a un modelo documental, requisitos adicionales como la integridad pueden justificar un enfoque relacional.

## 4.3. Caso 3: Comparación relacional vs. documental

En este experimento, se implementó un sistema de clientes y facturas (Cliente → Factura) en dos tecnologías: SQLite (relacional) y LiteDB (documental). Los resultados muestran que LiteDB requirió solo 151 líneas de código, frente a 454 en SQLite, una reducción del 66% en complejidad.



Figure 2 - Comparación de líneas de código en la asignación de permisos.[[4]](#footnote-4)

Como se observa en la imagen, la implementación basada en el modelo documental reduce la cantidad de código a un tercio del necesario con SQL recursivo.

**Análisis detallado:**

* **Consultas:** En SQLite, recuperar un cliente con sus facturas requiere JOINs; en LiteDB, basta con leer un solo documento.
* **Mantenimiento:** Modificar la estructura o añadir relaciones es más fácil en LiteDB, sin necesidad de alterar esquemas de tablas.
* **Rendimiento:** LiteDB destacó en operaciones de lectura, aunque en escrituras masivas el desempeño dependería de la implementación.

**Conclusión:** Para estructuras arbóreas, el modelo documental ofrece claras ventajas en simplicidad y rendimiento, especialmente en escenarios de lectura intensiva.

# 5. Discusión: Consecuencias estructurales y estratégicas del criterio

En la práctica habitual, la elección del modelo de persistencia suele hacerse al inicio del proyecto, con criterios tecnológicos o arbitrarios, y sin que medie un análisis estructural del dominio. Este trabajo propone una inversión de ese orden: primero se analiza la topología de las entidades y relaciones, y luego se elige el modelo de persistencia más natural. Este cambio de enfoque transforma una decisión arquitectónica en una consecuencia analítica, mejorando la coherencia del sistema.

## 5.1. Alineación con el proceso de desarrollo

Una de las principales fortalezas del criterio basado en topología es su **integración natural con las fases tradicionales del desarrollo de software orientado a objetos**, especialmente dentro de marcos estructurados como RUP simplificado. Esta alineación no es incidental, sino central para su aplicabilidad: el criterio se posiciona estratégicamente en la fase de diseño, como puente entre la comprensión del dominio (análisis) y su implementación técnica (programación).

**Fase de análisis: comprender el dominio y anticipar cambios**

Durante el análisis, el equipo identifica casos de uso, actores y procesos de negocio. En esta etapa se revelan las relaciones principales entre entidades, que constituyen la **topología lógica** del dominio. El criterio aprovecha esa información: no requiere modelos adicionales, sino que interpreta de manera estructural lo ya revelado por el análisis.

**Fase de diseño: elegir con criterio, no por inercia**

Aquí es donde el criterio se activa. Analizando si las relaciones forman **árboles** o **grafos**, se puede tomar una decisión informada sobre el modelo de persistencia más adecuado: jerárquico/documental o relacional/grafos, respectivamente. Esta decisión no solo evita ineficiencias posteriores (por ejemplo, JOINs innecesarios o estructuras artificiales), sino que también orienta la adecuación de las clases al modelo de persistencia con un propósito claro y justificado.

En otras palabras, el criterio se convierte en una **herramienta de diseño**, no una regla técnica post-implementación. Esto permite que las decisiones arquitectónicas surjan como consecuencia del análisis de la estructura del problema, no de la tradición tecnológica.

**Fase de programación: implementación coherente y sin distorsión**

Una vez definido el modelo de persistencia según la topología, la fase de programación puede enfocarse en traducir el diseño a código sin necesidad de adaptar artificialmente las clases del dominio a un esquema predefinido. En bases documentales, esto se traduce en serializaciones naturales; en modelos relacionales, en estructuras preparadas para manejar claves foráneas, transacciones o relaciones complejas.

El resultado es un código más limpio, expresivo y alineado con el diseño, porque la adecuación se hizo de forma anticipada y justificada, no reactiva.

**Reflexión final**

Al insertarse con precisión en la fase de diseño, el criterio topológico no interfiere con el flujo natural de trabajo, sino que **lo potencia**. Actúa como un **filtro de decisión racional** que preserva la claridad semántica del dominio y anticipa problemas comunes en la persistencia. Esta integración armónica lo convierte en una herramienta poderosa para equipos que deseen construir sistemas sólidos, eficientes y adaptables desde el inicio.

## 5.2. Justificación para arquitecturas híbridas

Una virtud clave del criterio basado en topología es que **no exige una única solución para todo el sistema**, sino que permite evaluar **módulos de forma independiente**, habilitando así arquitecturas híbridas. Este enfoque modular es especialmente útil en sistemas complejos donde distintos componentes presentan **topologías lógicas distintas** o **requisitos técnicos divergentes**.

**Evaluación modular de topologías**

El criterio no se aplica de forma global, sino puntual. Cada módulo o agregado del sistema puede ser analizado individualmente según la estructura de sus entidades y relaciones. Por ejemplo:

* Un módulo de facturación con una estructura Cliente → Facturas → Ítems forma un **árbol jerárquico**, ideal para un modelo documental como MongoDB o LiteDB.
* Un módulo de cuentas corrientes con relaciones entre Clientes ↔ Facturas ↔ Cobranzas configura un **grafo relacional**, mejor resuelto mediante una base como SQL Server o Neo4j.

Este análisis modular permite elegir el modelo de persistencia más adecuado **sin imponer un esquema único** para todo el sistema, manteniendo la coherencia topológica y maximizando la eficiencia de cada parte.

**Arquitectura flexible y desacoplada**

Para que esta estrategia sea viable, la arquitectura del sistema debe diseñarse con principios de desacoplamiento: clases bien definidas, capas independientes, interfaces claras para el acceso a datos y separación entre lógica de dominio y persistencia. Bajo este esquema, distintos repositorios pueden implementar distintas tecnologías sin afectar la lógica central del sistema.

Por ejemplo, una aplicación podría tener un IClienteRepository implementado con LiteDB, y un ICuentaCorrienteRepository con SQLite, ambos inyectados a través de interfaces y completamente transparentes para el resto del sistema. Esto **preserva la modularidad** y permite evolucionar los módulos de forma autónoma si cambian sus requisitos.

**Límites técnicos y decisiones arquitectónicas**

Si bien las arquitecturas híbridas ofrecen flexibilidad, no siempre son viables. **Requisitos como integridad referencial global, transacciones cruzadas o gran escala** pueden imponer restricciones:

* Si se requiere consistencia fuerte entre módulos con diferentes topologías (por ejemplo, facturación y cobranzas), mantener integridad en distintos motores puede volverse inviable.
* En sistemas con miles de relaciones cruzadas, un único modelo relacional puede ser preferible por la robustez de sus garantías ACID.

El criterio **no niega estas limitaciones**, pero permite detectarlas con antelación. Gracias a su aplicación en la fase de diseño, el equipo puede prever dónde usar una persistencia híbrida y dónde unificar.

**Conclusión**

Las arquitecturas híbridas dejan de ser decisiones arbitrarias o emergentes, y se convierten en **una consecuencia lógica** del análisis topológico del dominio. El criterio habilita esta flexibilidad con fundamentos estructurales, evitando elecciones por moda o costumbre. Cuando se aplica correctamente, cada módulo usa el modelo de persistencia que mejor se adapta a su forma interna, garantizando así un sistema más eficiente, claro y evolutivo.

## 5.3. Mitos relacionales y su desmontaje

La persistencia relacional ha sido durante décadas el estándar dominante en el desarrollo de software. Su madurez, estabilidad y amplio soporte la convierten en una elección razonable en muchos escenarios. Sin embargo, esa hegemonía ha generado una **serie de suposiciones automáticas** que rara vez se cuestionan, incluso cuando la topología del dominio sugiere modelos alternativos más simples y eficientes.

Este criterio no pretende invalidar el modelo relacional, sino **invitar a una evaluación más racional y topológicamente informada**, desmitificando algunos supuestos comunes.

**Mito 1: “Lo relacional es siempre más seguro”**

Se suele asociar a las bases relacionales con mayor seguridad, debido a su soporte completo para propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad). Si bien estas garantías son reales y valiosas, **no son exclusivas del modelo relacional**.

Bases documentales modernas como MongoDB han incorporado soporte ACID en operaciones transaccionales dentro de colecciones (e incluso entre ellas, desde la versión 4.0). Para topologías arbóreas, donde los datos de una entidad y sus hijos están contenidos en un único documento, estas transacciones son innecesarias: la escritura de un documento ya es atómica por naturaleza.

**Conclusión:** la seguridad transaccional puede obtenerse también en modelos documentales, siempre que la topología lo permita.

**Mito 2: “Lo relacional sirve para todo”**

El modelo relacional fue diseñado para representar conjuntos planos de datos con relaciones explícitas entre tablas. Esta estructura es ideal para topologías de grafo, pero **introduce complejidad artificial cuando se aplica a árboles jerárquicos**, típicos en sistemas orientados a objetos.

Por ejemplo, modelar un cliente con sus facturas e ítems asociados en SQL requiere múltiples tablas, claves foráneas y JOINs complejos. En contraste, una base documental permite almacenar toda la jerarquía en un único documento, directamente alineado con la estructura de clases en C#. Esto reduce el esfuerzo de mapeo y simplifica las consultas.

**Conclusión:** no hay un modelo único que sirva para todo; el modelo debe elegirse según la forma de los datos.

**Mito 3: “Las estructuras relacionales son más mantenibles”**

Esta creencia parte de la idea de que las bases relacionales son “más claras” o “más formales”. Sin embargo, cuando se fuerza una estructura jerárquica dentro de tablas planas, lo que se obtiene es un **esquema artificialmente fragmentado**, que requiere reconstrucción constante (por ejemplo, mediante CTEs) y lógica interpretativa en el código.

Ejemplo concreto: el uso de string[] AutosId en una clase Persona para relaciones optativas y de baja cardinalidad (menos de 20 autos) puede ser más expresivo y mantenible que una tabla intermedia con claves foráneas. El esquema resultante es más simple, más alineado con el dominio, y más fácil de evolucionar.

**Conclusión:** la mantenibilidad depende de la coherencia entre la estructura del dominio y el modelo de persistencia, no del tipo de base.

**Reflexión final**

Cuestionar la tradición relacional no implica descartarla, sino devolverle su lugar: **una herramienta útil, pero no universal**. El criterio basado en topología invita a mirar el diseño con ojos estructurales: ¿cómo son las relaciones?, ¿qué forma tienen los datos?, ¿qué modelo las expresa con mayor fidelidad y eficiencia?

Al desmontar estos mitos con evidencia y sentido común, se abre espacio para tomar decisiones **menos dogmáticas y más racionales**, alineadas con el paradigma orientado a objetos y el principio de claridad del dominio.

## 5.4. Objeciones frecuentes y respuestas fundadas

**¿Y si el usuario necesita buscar por un dato que está dentro del array embebido (por ejemplo, AutosId)?**

**Respuesta:** Las bases documentales modernas permiten buscar dentro de arrays sin problemas. Por ejemplo, MongoDB soporta queries por elementos embebidos ($elemMatch) y búsquedas directas por valor. LiteDB también permite búsquedas en colecciones dentro de documentos. Esta capacidad hace innecesaria la normalización para búsquedas simples en relaciones de baja cardinalidad.

**¿Y si más adelante necesitamos saber a qué personas pertenece un determinado auto?**

**Respuesta:** En ese caso, pueden indexarse los AutosId o denormalizar la información según necesidad. Alternativamente, si la relación cambia de naturaleza (pasa de optativa a central), se puede refactorizar el modelo sin migrar toda la arquitectura: el criterio no impide refactorizar, pero evita complejidades innecesarias desde el inicio. Diseñar con lo que es ahora, no con lo que podría ser, es una práctica sensata.

**¿Y si necesito agregar o eliminar autos sin tocar la persona entera?**

**Respuesta:** Se puede usar $push, $pull o equivalentes para modificar arrays embebidos sin reescribir el documento completo. Además, si la mutabilidad de la colección es frecuente, eso es una señal topológica de que la relación debería ser externa, y el criterio lo admitiría como caso de estructura más compleja (grafo). Nuevamente, el criterio no impone un modelo, lo descubre.

**¿Qué pasa con la integridad referencial?**

**Respuesta:** En relaciones duras y críticas, como Factura → Cliente, un modelo relacional es adecuado. Pero en relaciones blandas, optativas, o de baja cardinalidad, el costo de asegurar integridad a nivel de motor puede superar sus beneficios. El criterio propone que la integridad referencial no debe ser dogma, sino decisión de diseño contextualizada.

**¿Y si mañana cambiamos de modelo? ¿Cómo migramos los datos?**

**Respuesta:** La migración es siempre un desafío, sin importar el modelo. Pero el uso de una arquitectura bien desacoplada, con repositorios e interfaces claras, permite encapsular el acceso a datos y minimizar el impacto del cambio. El problema no está en usar un modelo u otro, sino en acoplarse rígidamente a él.

**Conclusión**

Estas objeciones no invalidan el criterio; al contrario, lo enriquecen. Su respuesta muestra que el criterio es flexible, adaptativo y diseñado para contextos reales, no ideales. No parte de la negación del modelo relacional, sino de su integración crítica dentro de un ecosistema donde la forma de los datos guía las decisiones, no la tradición.

## 5.5. Diseño orientado a principios

El criterio topológico no solo resuelve problemas técnicos de persistencia, sino que se alinea profundamente con los **principios fundamentales del diseño orientado a objetos**, en particular con los principios SOLID. Esta coherencia no es accidental: refleja un enfoque que prioriza la **expresividad del dominio, la claridad estructural y la evolutividad del sistema**.

**Principio de Responsabilidad Única (SRP)**

Cuando una clase Persona contiene un arreglo de identificadores de autos (string[] AutosId), está expresando que posee una relación opcional con esos autos, pero sin responsabilizarse de su existencia ni su gestión interna. Si se forzara a una tabla intermedia, se estaría trasladando a otra estructura una responsabilidad que es naturalmente parte del objeto persona.

El criterio permite que cada clase exprese solo lo que le compete, sin asumir la lógica de estructura externa salvo que la topología lo exija.

**Principio de Abierto/Cerrado (OCP)**

Diseñar la persistencia en función de la topología del dominio permite que la estructura se mantenga abierta a extensión (nuevos tipos de relación o persistencia), pero cerrada a modificaciones innecesarias. Si una relación cambia de naturaleza (por ejemplo, de embebida a relacional), el cambio se puede encapsular a nivel de repositorio, sin reestructurar las clases de dominio.

**Principio de Sustitución de Liskov (LSP)**

El criterio favorece la coherencia estructural: un Usuario con o sin permisos especiales puede seguir siendo tratado como Usuario, sin violar su contrato de comportamiento. Esta estabilidad semántica es clave para que las estructuras jerárquicas no introduzcan excepciones ocultas ni casos límite que violen LSP.

**Principio de Segregación de Interfaces (ISP)**

Aplicar el criterio implica usar interfaces específicas para cada forma de persistencia. Un repositorio que gestiona objetos con relaciones embebidas puede tener una interfaz distinta de uno que maneja claves foráneas o transacciones. Esto evita la creación de repositorios genéricos sobrecargados y favorece la modularidad.

**Principio de Inversión de Dependencias (DIP)**

Finalmente, al estructurar el acceso a datos mediante interfaces (por ejemplo, IUserRepository o ICarRepository), el sistema invierte la dependencia de los detalles técnicos: la lógica de negocio no sabe ni le importa si los datos se almacenan en SQLite, LiteDB o JSON. Esa decisión está aislada y responde únicamente a la forma de los datos.

**Reflexión final**

Aplicar este criterio no es un acto de rebeldía tecnológica, sino de fidelidad al espíritu del diseño orientado a objetos: dejar que la forma del dominio determine la forma del sistema, y no al revés. En ese sentido, la topología es una verdad estructural, y honrarla con decisiones de persistencia coherentes es una forma madura de hacer diseño.

Cuando cada parte del sistema conoce su forma y su rol, cuando la lógica no se pliega a la herramienta, sino que la elige con criterio, entonces se construyen sistemas sólidos, claros y humanos.

Un diseño que alinea la estructura lógica con la física no solo es más claro y directo, sino también más fácil de extender y mantener. Al reducir el código redundante y eliminar adaptaciones forzadas, se disminuye la carga cognitiva sobre el equipo de desarrollo. En este sentido, el criterio propuesto también puede entenderse como una inversión en calidad de software a mediano y largo plazo.

# 6. Conclusión: un criterio estructural para una decisión estructurante

Este trabajo propuso un criterio simple, pero profundamente estructural: **elegir el modelo de persistencia según la topología lógica del dominio**. A lo largo del desarrollo, se mostró que este criterio no solo es viable en la práctica, sino que además:

* Se integra naturalmente con las fases tradicionales del desarrollo orientado a objetos.
* Permite modularidad y adaptación mediante arquitecturas híbridas.
* Evita errores clásicos del modelo relacional cuando se aplica fuera de su territorio natural.
* Resiste objeciones técnicas frecuentes con fundamentos claros.
* Se alinea con los principios SOLID y promueve un diseño expresivo y mantenible.

El punto no es elegir entre SQL o NoSQL, entre lo clásico o lo moderno. El punto es **respetar la forma del dominio**, y dejar que esa forma nos hable, nos sugiera, nos guíe. En un mundo donde las decisiones arquitectónicas muchas veces se toman por hábito, moda o presión externa, este criterio devuelve al diseño su lugar central como **acto racional, estructural y humano**.

Este paper no busca ser una verdad cerrada, sino **una apertura**: una forma de ver que hay decisiones que pueden tomarse mejor, con más conciencia, más claridad, y sobre todo, más respeto por lo que el software intenta modelar. Es decir, este criterio no solo optimiza sistemas, sino que empodera a los diseñadores a escuchar la voz del dominio y construir software que respete su esencia.

Los aportes concretos de este trabajo son:

1. Un criterio práctico y reproducible para elegir el modelo de persistencia según la topología lógica de las entidades.
2. Una validación empírica basada en implementaciones contrastadas con dos tecnologías reales (LiteDB y SQLite).
3. La recuperación del diseño estructural como eje articulador entre análisis, diseño e implementación.
4. Un ejemplo de cómo integrar teoría y práctica sin necesidad de sofisticación innecesaria.

El aporte principal de este trabajo no reside en la implementación concreta, sino en el criterio mismo. Se trata de una herramienta conceptual simple, pero poderosa, que puede incorporarse a cualquier proceso de diseño orientado a objetos. Su aplicación permite tomar decisiones de persistencia informadas, coherentes con la estructura del dominio, y evita errores comunes que suelen arrastrarse durante todo el ciclo de vida del software.

# 7. Referencias

La hipótesis del criterio basado en topología presentada en este artículo se fundamenta en una combinación de casos prácticos desarrollados para ilustrar su aplicación, junto con conceptos extraídos de literatura relevante y estándares establecidos en el diseño de software. Aunque no se basa en una tesis específica, los conceptos citados proporcionan el contexto teórico y práctico para la crítica a la tradición relacional y la promoción de modelos no relacionales.

**Libros y textos clásicos**

1. **Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.** Este libro es un pilar del diseño orientado a objetos y habla de patrones que aseguran flexibilidad y modularidad. El criterio topológico se beneficia de estas ideas al buscar modelos de persistencia que no rompan la lógica del dominio.
2. **Martin, R. C. (2003). Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.** En este libro se explican los principios SOLID, que son esenciales para que un diseño sea claro y fácil de mantener. El enfoque topológico encaja con los conceptos de responsabilidad única y desacoplamiento.
3. **Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley.** Fowler explora patrones de persistencia como el ORM y discute la "impedancia" entre objetos y bases relacionales. Esto conecta directamente con el análisis sobre cuándo usar bases NoSQL o relacionales según la estructura de datos.

**Artículos y estudios sobre bases de datos**

1. **Stonebraker, M., & Moore, D. (1996). Object-Relational DBMSs: The Next Great Wave. Morgan Kaufmann.** Este texto muestra las limitaciones de las bases relacionales para datos complejos, lo que apoya la idea de buscar alternativas como NoSQL para ciertas topologías.
2. **Sadler, C. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley.** Este texto trata sobre la "persistencia políglota", que es inherente a esta propuesta de adaptar el modelo de persistencia a la topología de cada módulo.
3. **Codd, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM, 13(6), 377-387.** Este texto dio origen al modelo relacional, modelo que este trabajo no descarta sino que lo señala como opción válida para topologías específicas (como grafos).

**Investigaciones sobre topología y estructuras de datos**

1. **Harary, F. (1969). Graph Theory. Addison-Wesley.** Este clásico de teoría de grafos es el respaldo teórico para hablar de árboles y grafos en el criterio topológico.
2. **Chen, P. P. (1976). "The Entity-Relationship Model—Toward a Unified View of Data". ACM Transactions on Database Systems, 1(1), 9-36.** El modelo ER es clave en bases relacionales y esta propuesta lo complementa al ofrecer una visión más flexible basada en topología.

**Trabajos sobre persistencia en sistemas orientados a objetos**

1. **Atkinson, M. P., & Buneman, O. P. (1987). "Types and Persistence in Database Programming Languages". ACM Computing Surveys, 19(2), 105-170.** Este artículo aborda los retos de la persistencia en lenguajes orientados a objetos y apoya la idea de alinear clases con modelos de almacenamiento.
2. **Maier, D. (2001). "Database Desiderata for an XML World". Proceedings of the 17th International Conference on Data Engineering, 561-570.** Explora cómo XML (y por extensión, modelos documentales) maneja datos jerárquicos, lo que refuerza los ejemplos prácticos con estructuras arbóreas.

**Estudios de caso y comparativas**

1. **Parker, Z., Poe, S., & Vrbsky, S. V. (2013). "Comparing NoSQL MongoDB to an SQL Database". Proceedings of the 51st ACM Southeast Conference, 1-6.** Compara MongoDB y bases SQL, dando datos concretos que justifican elegir modelos documentales para ciertas topologías.
2. **Stonebraker, M., & Cattell, R. (2011). "10 Rules for Scalable Performance in Simple Operation Datastores". Communications of the ACM, 54(6), 72-80.** Analiza cómo las bases relacionales fallan en escalabilidad, lo que respalda el énfasis en considerar la escala al elegir un modelo de persistencia.

1. **PlantUML:**

   @startjson

   {

   "Id": "C001",

   "Nombre": "Cliente Ejemplo",

   "Facturas": [

   {

   "IdFactura": "F001",

   "Fecha": "2025-04-20",

   "Items": [

   { "IdItem": "I001", "Descripcion": "Producto A" },

   { "IdItem": "I002", "Descripcion": "Producto B" }

   ]

   }

   ]

   }

   @endjson [↑](#footnote-ref-1)
2. @startuml

   start

   :Análisis OO;

   :Topología lógica;

   :Elección del modelo de persistencia;

   :Diseño OO (estructuras, entidades);

   :Implementación OO (C#, LiteDB/SQLite);

   stop

   @enduml [↑](#footnote-ref-2)
3. @startuml

   start

   :Análisis de relaciones lógicas entre entidades;

   if (¿Forman un árbol?) then (Sí)

   :Usar modelo documental;

   else (No)

   if (¿Forman un grafo?) then (Sí)

   :Usar modelo relacional;

   else (No)

   :Caso no cubierto – analizar estructura específica;

   endif

   endif

   stop

   @enduml [↑](#footnote-ref-3)
4. @startgantt

   [SQLite] requires 45 days

   [SQLite] is 0% complete

   [LiteDB] requires 15 days

   [LiteDB] is 0% completed

   @endgantt [↑](#footnote-ref-4)