RESUMEN DEL LIBRO DE PERSISTENCIA

# Capítulo I: Persistencia de Objetos

**Características deseables de la persistencia orientada a objetos**

* **Performance sin compromisos:** Equilibrar diseño y eficiencia.
* **Transparencia:** El desarrollador no necesita preocuparse por la lógica de persistencia, solo por la lógica de negocio.
* **Ortogonalidad:** Separar las preocupaciones de diseño de dominio y persistencia. **un concepto A es ortogonal a otro concepto B si al pensar en A no es necesario tener presente B**
* **Uso de POJO (Plain Old Java Objects):** Clases simples sin dependencias innecesarias en tecnologías de persistencia.
* **Persistencia por alcance:** Todos los objetos accesibles desde un objeto persistente se almacenan automáticamente. Para persistir objetos es necesario solamente vincularlos con algún objeto ya persistente. (Ejemplo: si tengo un objeto Persona que ya esta persistido y quiero persistir un objeto Telefono, vinculo a teléfono con persona)

**Operaciones CRUD**

* Proporciona métodos intuitivos para crear, leer, actualizar y eliminar objetos en una base de datos sin necesidad de realizar ajustes complejos en el diseño del dominio.

1. **C** (Create): Encargada de crear nueva información persistente. → no debería existir una operación "create" o "save" para almacenar instancias en el repositorio.
2. **R** (Retrieve, Read):Encargada de recuperar la información del modelo.
3. **U** (Update):para actualizar un objeto basta con enviar un mensaje (setter) con la información nueva a actualizar.  → no es necesario tener operaciones "update" diferenciadas para actualizar la información.
4. **D** (Delete): para borrar un objeto del modelo lo único que debería hacerse es "desvincularlo" apropiadamente de sus colaboradores

# Capítulo II: Ambientes Concurrentes y Transacciones

**Propiedades ACID**

* **Atomicidad:** Una transacción es completamente exitosa o se revierte.
* **Consistencia:** El estado de la base de datos debe mantenerse válido antes y después de la transacción.
* **Aislamiento:** Las transacciones concurrentes no interfieren entre sí.
* **Durabilidad:** Los cambios confirmados permanecen incluso en caso de fallos del sistema.

**Esquemas de bloqueo**

* **Optimista:** Asume que los conflictos son raros; las transacciones fallan al intentar confirmar cambios si hay conflictos. Evita inanición
* **Pesimista:** Bloquea recursos durante la transacción, garantizando exclusividad a costa de rendimiento.

**Versionamiento**

* Uso de múltiples versiones de los datos para evitar conflictos de escritura y lectura en escenarios concurrentes.

# Capítulo III: Bases de Datos Orientadas a Objetos (BDOOs)

**Definición y características**

* Las BDOOs son colecciones de objetos persistentes que soportan las propiedades ACID.
* Representan un modelo más natural para aplicaciones orientadas a objetos, eliminando las discrepancias entre el modelo conceptual y la implementación.

**Ventajas de las BDOOs**

* Diseños más intuitivos y menos código de transformación.
* Mejora en la performance de consultas relacionadas con datos complejos.
* Ejemplos de aplicación incluyen sistemas en industrias como la física de partículas y vigilancia.

**Coexistencia con bases relacionales**

* Se explora cómo bases de datos orientadas a objetos y relacionales pueden coexistir para cumplir diferentes propósitos en sistemas híbridos.

# Capítulo IV: Mapeo Objeto-Relacional (ORM)

El capítulo detalla el desafío de integrar el mundo de los objetos con el modelo relacional, superando la disparidad entre ambos paradigmas.

**El Problema del Mapeo Objeto-Relacional**

* Los sistemas orientados a objetos usan jerarquías de clases y referencias directas entre objetos.
* Las bases de datos relacionales emplean tablas, filas y columnas, lo que introduce una "disonancia" en la representación de datos.
* Resolver esta brecha requiere herramientas y estrategias específicas que preserven la estructura del diseño orientado a objetos.

**Estrategias de Mapeo para Jerarquías de Clases**

1. **Una tabla por jerarquía:**
   * Todas las clases de una jerarquía comparten una única tabla en la base de datos.
   * **Ventajas:**
     + Fácil de implementar y rápida de consultar.
   * **Desventajas:**
     + La tabla puede crecer significativamente si la jerarquía es grande.
     + Puede haber muchas columnas nulas para atributos no aplicables a todas las clases.

**Ejemplo:**

Tabla: Vehiculo

1. id | tipo | velocidad\_max | num\_ruedas | combustible

2. --------------------------------------------------------

3. 1 | auto | 200 | 4 | gasolina

4. 2 | moto | 150 | 2 | gasolina

1. **Una tabla por clase concreta:**
   * Cada clase concreta en la jerarquía tiene su propia tabla con todos sus atributos.
   * **Ventajas:**
     + Los datos están completamente normalizados.
   * **Desventajas:**
     + Consultas que involucran la jerarquía completa son complejas.

**Ejemplo:**

1. Tabla: Auto

2. id | velocidad\_max | combustible

3. -----------------------------------

4. 1 | 200 | gasolina

1. **Una tabla por clase (herencia por tabla):**
   * Cada clase (incluidas las abstractas) tiene su tabla. Las subclases contienen columnas adicionales y claves foráneas que apuntan a la tabla padre.
   * **Ventajas:**
     + Consultas más eficientes para subclases específicas.
   * **Desventajas:**
     + Mayor complejidad en el diseño y las consultas.

**Ejemplo:**

1. Tabla: Vehiculo

2. id | tipo

3. ---------------

4. 1 | auto

5.

1. Tabla: Auto

2. id | velocidad\_max | combustible

3. -----------------------------------

4. 1 | 200 | gasolina

5.

**Herramientas de ORM**

1. **Hibernate:**
   * Un framework ampliamente utilizado en Java para manejar la persistencia.
   * Características destacadas:
     + Uso de HQL (Hibernate Query Language) para realizar consultas.
     + Gestión automática de transacciones y cache.
     + Soporte para herencia, colecciones y asociaciones.
2. **JPA (Java Persistence API):**
   * Estándar de Java para ORM que abstrae el mapeo entre objetos y tablas relacionales.
   * Utiliza anotaciones en las clases Java para definir el mapeo.

# Capítulo V: Patrones de Diseño para Persistencia

Este capítulo describe los patrones de diseño más comunes que facilitan la separación de responsabilidades, mejoran la modularidad y simplifican la persistencia en aplicaciones orientadas a objetos.

**Patrones de Diseño Importantes**

1. **DTO (Data Transfer Object):**
   * Su propósito es transferir datos entre capas sin exponer los detalles internos de las entidades de dominio.
   * **Ventajas:**
     + Mejora la encapsulación.
     + Reduce el tráfico de red al transportar solo los datos necesarios.
   * **Ejemplo:**

1. class ClienteDTO {

2. private String nombre;

3. private String email;

4.

5. // Getters y setters

6. }

1. **Repositorio:**
   * Actúa como un intermediario entre el dominio y la capa de persistencia.
   * Centraliza la lógica de acceso a datos y mejora la cohesión.
   * **Ejemplo:**

1. interface ClienteRepositorio {

2. Cliente buscarPorId(int id);

3. List<Cliente> buscarTodos();

4. }

1. **Decorador:**
   * Permite añadir funcionalidad a los objetos persistentes sin modificar su estructura original.
   * **Ventajas:**
     + Promueve la reutilización de código.
     + Añade funcionalidades dinámicamente.

**Herramientas para Patrones**

1. **Spring Data:**
   * Framework que simplifica la creación de repositorios basados en interfaces.
   * Soporte para consultas personalizadas y generación automática de métodos.
2. **JDO (Java Data Objects):**
   * Abstracción para manejar la persistencia sin preocuparse por el tipo de base de datos utilizada.
   * Soporta tanto bases de datos relacionales como no relacionales.

CONCEPTOS:

Un sistema legacy o sistema heredado es un software antiguo que sigue siendo utilizado dentro de una organización debido a su importancia crítica para las operaciones, a pesar de que puede estar basado en tecnologías obsoletas o haber sido reemplazado parcialmente por soluciones más modernas​​.

Repositorio: Abstrae la lógica de persistencia, proporcionando una interfaz uniforme para acceder a los datos.

DTO (Data Transfer Objects): Facilitan la transferencia de datos entre capas, mejorando la separación de responsabilidades

JDO (Java Data Objects):Abstracción para manejar la persistencia sin preocuparse por el tipo de base de datos utilizada.

EXPLICACIÓN DEL POWER POINT

**Características de los sistemas legacy:**

1. **Dependencia tecnológica**:
   * Usan tecnologías o lenguajes de programación antiguos, como COBOL, que tienen soporte limitado en la actualidad.
   * Pueden carecer de documentación completa.
2. **Importancia crítica**:
   * Están profundamente integrados en los procesos de negocio y suelen ser responsables de funciones esenciales como la gestión financiera, inventarios o recursos humanos.
3. **Mantenimiento complicado**:
   * Su actualización es costosa y riesgosa debido a su antigüedad y a la falta de expertos familiarizados con su tecnología.
4. **Dificultad de integración**:
   * No suelen estar diseñados para interoperar con tecnologías modernas, lo que puede generar problemas al intentar integrarlos con nuevos sistemas.

**Ventajas:**

* **Estabilidad**: Estos sistemas suelen ser confiables porque han sido probados exhaustivamente a lo largo del tiempo.
* **Costos hundidos**: La inversión inicial ya se realizó, lo que reduce los costos a corto plazo.

**Desventajas:**

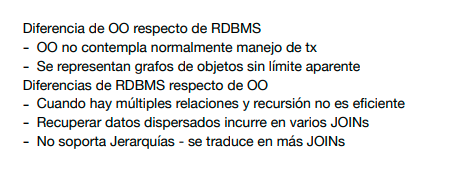
* **Altos costos de mantenimiento**: Mantener estos sistemas es caro, tanto en tiempo como en recursos.
* **Falta de flexibilidad**: Adaptarlos a nuevos requerimientos es complicado y a menudo limitado.
* **Riesgo de fallos críticos**: Su antigüedad los hace vulnerables a errores que no siempre pueden solucionarse fácilmente.

**Reemplazo o modernización**

Organizaciones que utilizan sistemas legacy enfrentan un dilema entre:

1. **Modernizar**: Actualizar los sistemas para mejorar su rendimiento y adaptarlos a nuevas necesidades.
2. **Reemplazar**: Sustituirlos por soluciones más modernas, aunque esto implica un alto costo inicial y riesgos en la transición.
3. **Conviviendo con ambos**: Usar capas de integración (middleware) para que los sistemas legacy interactúen con tecnologías nuevas.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



**Diferencias de OO respecto de RDBMS**

**1. OO no contempla normalmente manejo de transacciones (tx)**

* **Explicación**:
  + En el paradigma orientado a objetos (OO), las transacciones no son un concepto nativo. Los objetos se crean y destruyen en memoria, y su estado se gestiona a través de métodos encapsulados.
  + Las operaciones sobre los objetos no suelen incluir características transaccionales como commit, rollback o atomicidad por defecto.
* **Implicación**:
  + Para sistemas que requieren persistencia o consistencia transaccional, es necesario delegar este control a sistemas externos, como bases de datos relacionales.
  + Las herramientas ORM (Object-Relational Mapping), como Hibernate, agregan una capa que permite manejar transacciones al interactuar con bases de datos, pero esto no forma parte intrínseca del diseño orientado a objetos.

**2. Se representan grafos de objetos sin límite aparente**

* **Explicación**:
  + En OO, las relaciones entre objetos pueden formar estructuras complejas como grafos, en los que los objetos tienen referencias directas a otros objetos. Estas referencias pueden ser bidireccionales o incluir ciclos.
  + No existen restricciones explícitas como las claves foráneas en bases de datos relacionales, lo que da mayor flexibilidad para modelar relaciones complejas en memoria.
* **Implicación**:
  + Esta flexibilidad puede llevar a problemas de gestión de memoria y complejidad al momento de persistir estas estructuras en un sistema relacional.

**Diferencias de RDBMS respecto de OO**

**1. Cuando hay múltiples relaciones y recursión no es eficiente**

* **Explicación**:
  + En un modelo relacional, las relaciones entre entidades se manejan mediante claves primarias y foráneas. Aunque esto es eficiente para operaciones simples, cuando las relaciones son profundas o implican recursividad (como jerarquías de empleados o relaciones padres-hijos), las consultas SQL pueden volverse complicadas y lentas.
  + Ejemplo: Recuperar una jerarquía completa de empleados en una organización puede requerir múltiples consultas o un mecanismo especial como **CTE (Common Table Expressions)**.
* **Implicación**:
  + Las bases de datos relacionales no están diseñadas para manejar estructuras de datos profundamente interconectadas con la misma eficiencia que los modelos de objetos en memoria.

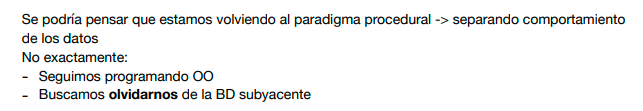
**2. Recuperar datos dispersos incurre en varios JOINs**

* **Explicación**:
  + Las bases de datos relacionales dependen de las operaciones JOIN para combinar datos de múltiples tablas relacionadas. Sin embargo, estas operaciones pueden ser costosas en términos de tiempo y recursos, especialmente cuando involucran muchas tablas o grandes volúmenes de datos.
* **Implicación**:
  + En comparación, en OO, los datos relacionados suelen estar directamente disponibles mediante referencias, eliminando la necesidad de operaciones adicionales para "unir" datos.

**3. No soporta jerarquías - se traduce en más JOINs**

* **Explicación**:
  + Las bases de datos relacionales no tienen un soporte nativo para jerarquías como las que existen en OO con herencia. Para representar jerarquías en un modelo relacional, se deben emplear estrategias como:
    1. **Tabla única por jerarquía**: Combina todas las clases en una sola tabla, pero introduce columnas adicionales para manejar atributos específicos de subclases.
    2. **Tabla por clase**: Crea una tabla por cada clase, pero requiere JOINs para reconstruir la jerarquía.
    3. **Tabla por subclase concreta**: Crea una tabla para cada subclase concreta, lo cual simplifica las consultas pero dificulta la extensibilidad.
* **Implicación**:
  + Esto introduce un esfuerzo adicional para modelar, consultar y mantener jerarquías, algo que es intrínseco y directo en OO.

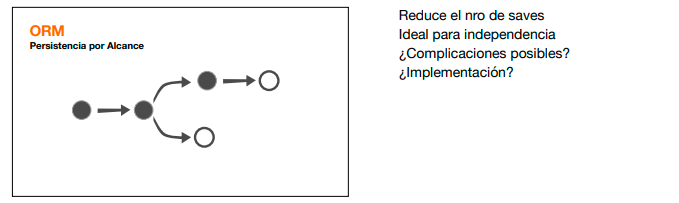
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



**Separación entre datos y comportamiento (Persistencia en OO vs Procedural)**

La persistencia en el paradigma orientado a objetos no implica volver al modelo procedural porque:

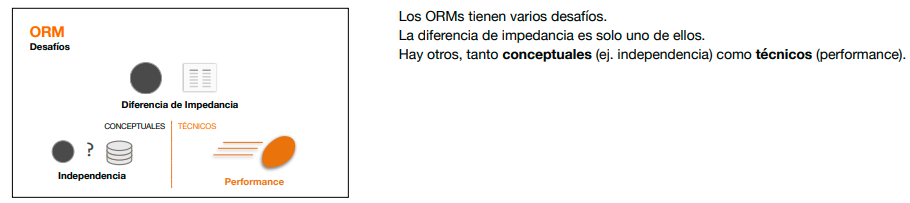
* **Persistencia transparente**: En OO, el diseño debe permitir que los desarrolladores se centren únicamente en el modelo de dominio sin preocuparse por los detalles de la base de datos subyacente. Esto se logra mediante mecanismos que ocultan la complejidad de la persistencia​.
* **Enfoque en los objetos**: La lógica de negocio y el comportamiento siguen estando dentro de los objetos, evitando romper el principio de encapsulación. Aunque el mapeo a una base de datos requiere adaptaciones, el objetivo es mantener un diseño limpio y OO​.



**Persistencia por Alcance**

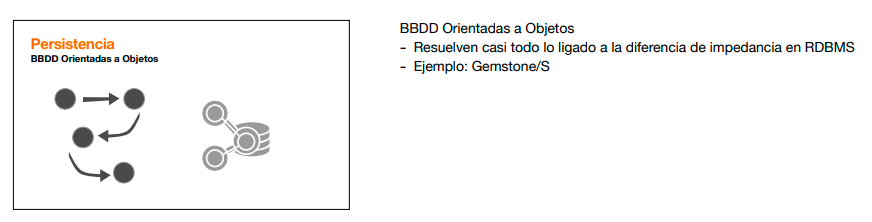
La persistencia por alcance establece que cualquier objeto navegable desde otro ya persistente también debe ser persistente. Ventajas y desafíos:

1. **Ventajas**:
   * Reduce la cantidad de operaciones explícitas de guardado (save), ya que la persistencia ocurre automáticamente a través de las relaciones.
   * Ideal para mantener la independencia del dominio respecto a la capa de persistencia​.
2. **Complicaciones posibles**:
   * Dificultad para controlar el alcance: Puede ser difícil determinar qué objetos se persisten automáticamente, especialmente en grafos complejos.
   * Problemas de rendimiento: Persistir automáticamente grandes grafos de objetos puede ser costoso en términos de rendimiento​.
3. **Implementación**:
   * Utilizar frameworks de ORM como Hibernate que admiten esta funcionalidad.
   * Diseñar relaciones de manera cuidadosa para evitar persistir objetos innecesarios o duplicados​​.



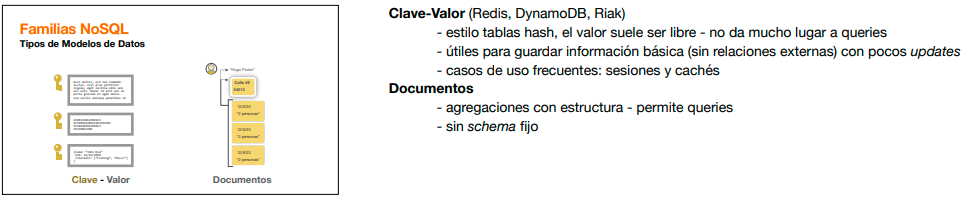
**Desafíos de los ORMs**

1. **Diferencia de impedancia**:
   * Surge porque los sistemas OO gestionan grafos de objetos, mientras que las bases de datos relacionales operan en tablas. Esto complica la representación de relaciones, jerarquías y consultas​​.
2. **Otros desafíos**:
   * **Independencia**: Es difícil lograr que el modelo de dominio no dependa del ORM, ya que los objetos pueden requerir anotaciones o configuraciones específicas del ORM.
   * **Performance**: Las consultas generadas por el ORM pueden no estar optimizadas, especialmente para casos complejos​.

****

**Bases de Datos Orientadas a Objetos (BDOO)**

1. **Beneficios**:
   * Resuelven problemas de diferencia de impedancia al trabajar directamente con objetos en lugar de tablas.
   * Ejemplo: Gemstone/S ofrece una persistencia nativa para objetos​.
2. **Consideraciones**:
   * A pesar de sus beneficios, las BDOOs no son tan comunes como las bases de datos relacionales debido a su menor adopción en la industria y la falta de estándares amplios​​.

****

**Clave-Valor y Documentos**

1. **Bases de datos clave-valor (Redis, DynamoDB, Riak)**:
   * Funcionan como grandes tablas hash: las claves identifican valores, que suelen ser datos sin estructura o con mínima estructura.
   * Usos comunes: Gestión de sesiones, cachés o datos que no requieren relaciones complejas ni actualizaciones frecuentes​​.
2. **Bases de datos de documentos**:
   * Almacenan datos en formatos como JSON o BSON con estructuras jerárquicas.
   * Permiten realizar consultas sobre los documentos y no requieren un esquema fijo.
   * Usos frecuentes: Sistemas que manejan datos semi-estructurados, como aplicaciones web o móviles​​.