Perfecto, Gerardo. Vamos a trabajar este artículo desde la raíz, con una **estructura clara** y enfocada en tu experiencia práctica y académica. Empecemos con un primer borrador de la **cadena de herencia**, y luego armamos el texto del artículo.

**🧱 Cadena base: Interfaz y clases para entidades auditables**

**1. Interfaz de auditoría – solo define los aspectos de registro**

public interface IAuditableEntity

{

DateTime CreatedAt { get; set; }

string CreatedBy { get; set; }

DateTime? ModifiedAt { get; set; }

string? ModifiedBy { get; set; }

bool IsDeleted { get; set; }

bool IsLocked { get; set; }

}

**2. Clase abstracta parametrizada con tipo de ID**

public abstract class AuditableEntity<TId> : IAuditableEntity

{

public TId Id { get; set; } // NO es descriptor, por eso está acá

public DateTime CreatedAt { get; set; }

public string CreatedBy { get; set; }

public DateTime? ModifiedAt { get; set; }

public string? ModifiedBy { get; set; }

public bool IsDeleted { get; set; }

public bool IsLocked { get; set; }

}

**3. Persistencia relacional (ej. Firebird)**

public abstract class RelationalEntity : AuditableEntity<int>

{

// Podés agregar metadata específica del modelo relacional si querés

}

**4. Persistencia documental (ej. RavenDB)**

public abstract class DocumentEntity : AuditableEntity<string>

{

// Metadata adicional de documentos (por ej. CollectionName)

}

**5. Entidad concreta del dominio**

public class Cliente : RelationalEntity // O DocumentEntity, según backend

{

public string Nombre { get; set; }

public string Documento { get; set; }

}

Con esto, podemos ahora armar el **artículo**. Acá va una propuesta de inicio, bien inspirada en lo que me contaste:

**📝 *Entidades Auditables y Descriptores de Clase: Integración Elegante entre Diseño y Persistencia***

**¿Qué es una entidad auditable?**  
Una entidad auditable es aquella que guarda trazabilidad sobre su ciclo de vida en un sistema: cuándo fue creada, por quién, cuándo fue modificada, si está bloqueada o eliminada lógicamente. En sistemas empresariales modernos, este patrón es vital no solo por cumplimiento normativo, sino también por robustez estructural.

**¿A qué viene este interés?**  
En mi tesis de grado, me encontré ante una restricción metodológica que me pareció, al inicio, arbitraria: las propiedades de una clase debían ser estrictamente *descriptoras* de la misma. En otras palabras, una clase Auto no podía tener una propiedad Id como descriptor, porque el identificador no describe al objeto, sino que lo referencia para la persistencia.

Este enfoque me obligó a segregar claramente entre lo que pertenece al modelo del dominio (los descriptores) y lo que pertenece al modelo de persistencia (identidad, timestamps, flags de borrado). En aquel momento, lo resolví con una clase EntidadPersistible de la cual heredaban las entidades del modelo. Hoy, entiendo que eso no solo era correcto, sino que corresponde a un patrón bien establecido: **el patrón de entidades auditables**.

**El desafío de la portabilidad entre modelos de persistencia**  
El problema se vuelve más interesante al considerar múltiples persistidores. ¿Cómo diseñar una jerarquía de clases que funcione igual para bases relacionales (como Firebird, con int autoincremental) y para bases documentales (como RavenDB, con string como clave)?

La solución está en **parametrizar la clase base con el tipo de ID**, dejando el modelo de dominio independiente de las decisiones de infraestructura. El ID deja de ser un descriptor y pasa a ser un aspecto técnico resuelto por herencia.

¿Te gustaría que siga con las secciones siguientes del artículo? Puedo continuar con:

* Ventajas del enfoque
* Ejemplo práctico
* Comparación con abordajes incorrectos
* Reflexión final sobre diseño limpio y flexible

¿Te gustaría también que armemos un pequeño diagrama UML de esta jerarquía para incluirlo visualmente en el artículo?