- El de fachada: representa el sistema global, dispositivo o subsistema, y se utiliza cuando no existen demasiados eventos del sistema o cuando no hay posibilidad de que la interfaz redireccione los distintos mensajes de los eventos a controladores alternativos.
- Controlador de casos de uso: existe un controlador diferente para cada caso de uso, aquí el controlador es una clase de fabricación pura que se utiliza para dar soporte al sistema.

Sin la clase controlador, es la interfaz quien debe conectarse con las clases de entidad y esa no es su responsabilidad. La responsabilidad de la interfaz es vincularse con el ambiente, No debería de tener metida la lógica de negocio (no debería saber cuál es el paso siguiente, sólo debería recibir la información del ambiente y pasarla al controlador, y que sea él quien tenga la responsabilidad de saber qué es lo que sigue).

WF DE ANÁLISIS

Tiene el protagonismo en las últimas iteraciones de la fase de inicio y en las primeras de la fase de elaboración.

De este wf sale el *modelo de análisis*, que es una primera aproximación al diseño, está escrito en el lenguaje del desarrollador y permite ver el sistema sin tener los detalles de la implementación, lo cual hace que sea mucho más fácil entender el sistema y tener un primer pantallazo de los que hace el mismo.

A diferencia del modelo de diseño, el modelo de análisis no resuelve todos los requerimientos y es genérico con respecto a la implementación.

En el wf de requerimientos uno determinaba los requerimientos, pero en el análisis es donde se especifican esos requerimientos y empezamos a abrir los casos de uso y no solamente ver lo que hacen, sino también ver cómo lo hacen (empezamos a pensar cómo vamos a programar esos casos de uso, pero sin especificar un lenguaje de programación; pensamos en una solución genérica que luego será especificada en el diseño).

La entrada del wf de análisis es el modelo de requerimientos (los casos de uso que se descubrieron).

Los trabajadores de wf de análisis son:

- Arquitecto: se encarga de mantener consistente y legible como un todo el modelo de análisis, también se encarga de refinar los paquetes y las clases que participan en las diferentes realizaciones de casos de uso-análisis.
- Ingeniero de casos de uso: se encarga de construir las realizaciones de casos de uso-análisis, respectando las clases participantes.
- Ingeniero de componentes: se encarga de mantener consistentes los paquetes y las clases que participan en los distintos casos de uso-análisis.

UNIDAD 2

WF DE DISEÑO

Sus salidas son el modelo de diseño y el modelo de despliegue; y tiene como entradas los requerimientos del wf de requerimientos y el modelo de análisis del wf de análisis.

En el diseño construimos un modelo físico que ya no es genérico con respecto a la implementación, sino que es específico.

Los trabajadores siguen siendo los mismos que en el análisis. Acá resolvemos los requerimientos no funcionales, aunque podemos posponer algunos para el momento de la implementación.

Acá se diseña la arquitectura y es por este motivo que este modelo debe ser mantenido a lo largo de todo el ciclo de vida del producto, a diferencia del análisis.

No modelamos la solución en términos lógicos, sino en términos físicos; se ocupa de la solución de todos los aspectos que tienen que ver con la implementación.

Tiene un papel preponderante en las últimas iteraciones de la fase de elaboración y en las primeras iteraciones de la fase de construcción.

Modelo de Diseño: Es un modelo de objetos que describe la realización física de los casos de uso, centrando la atención en como los RF y los RNF, junto con otras restricciones de implementación, tienen impacto en el sistema a considerar

Modelo de Despliegue: Es un modelo de objetos que describe la distribución física del sistema en cuanto a la distribución de la funcionalidad en los nodos de cómputo. Puede describir diferentes configuraciones de red. Representa una correspondencia entre la arquitectura del sw y la arquitectura del hw. Es la descripción de la arquitectura.

PRINCIPIOS DE DISEÑO

Características del buen diseño

- Identificación y tratamiento de excepciones: debemos permitirle al usuario que, frente a una situación de error, o acciones inadecuadas, o una excepción, él sepa cómo tiene que seguir a partir de ahí, de tal manera que pueda recuperarse del mismo.
- Prevención y tolerancia de defectos: debemos realizar un software que contemple si algo sale de lo normal.
- Independencia de componentes: se trata que los componentes sean independientes unos de otros. Se busca obtener un bajo acoplamiento y una alta cohesión.

PRINCIPIOS DE DISEÑO

Son una guía de las buenas prácticas que se deben seguir en el diseño de software, no siempre se pueden cumplir todos, pero la idea de estos principios es que sirvan como guía para generar un software mucho más mantenible, extensible y flexible.

Es una técnica que se aplica para diseñar o construir software, para hacerlo más flexible, extensible y mantenible.

Son guías de alto nivel que se aplican a cualquier lenguaje de programación OO. Su foco principal es mantener la cohesión alta, el acoplamiento bajo, facilitar los cambios y prevenir errores inesperados.

Son importantes porque son una guía para determinar si estamos generando un software de calidad o no.

Acoplamiento: nivel de dependencia entre clases. Que tanto afecta a una clase que otras cambien.
Cohesión: nivel de relación que existe entre los métodos y atributos que tiene una clase. Que una clase sea cohesiva significa que hace pocas cosas y que esas pocas cosas que se hace se encuentran altamente relacionadas.
Delegación: es cuando una clase le pasa la responsabilidad a otra de hacer una determinada acción, y para ello, le pasa todos los datos necesarios para que la pueda ejecutar, muchas veces incluso se pasa a ella misma como parámetro. (Lo aplican los patrones State y Strategy; está relacionado al principio TDA).

- ***DRY – Don't Repeat Yourself*** (State – Strategy)
  Promueve la reducción de la duplicación, especialmente en programación.
  Según este principio, toda pieza de información nunca debería ser duplicada, ya que la duplicación incrementa la dificultad en los cambios y evolución posterior.
  Hay que apuntar a que cada requerimiento esté en un único lugar, representado por una única porción de código.
  Mejora la mantenibilidad (los ajustes los tengamos que hacer en un solo lugar). Permite crear un código más legible y fácil de entender. Permite la reusabilidad. Mejora el testeo del software; y nos da una mayor velocidad en el desarrollo.

- ***TDA – Tell, Don't Ask*** (State)
  Se apunta a una distribución de responsabilidades equitativa.
  Debemos decirles a los objetos que hagan cosas, y estos objetos internamente tomarán sus propias decisiones según su estado.
  Un objeto le pide a otro que haga algo que necesita y que le devuelva el resultado.
- ***PTI – Program to Interface*** (Observer – Strategy)
  La herencia es un mecanismo para extender la funcionalidad de una aplicación reutilizando la funcionalidad de las clases padre. Permite definir un nuevo tipo de objetos basándose en otro, y obtener

así nuevas implementaciones casi sin esfuerzo, al heredar la mayoría de lo que se necesita, de clases ya existentes.
Todas las clases que derivan de una clase abstracta compartirán su interfaz. Esto implica que una subclase simplemente añade o redefine operaciones, y no oculta operaciones de la clase padre. Todas las subclases pueden responder entonces a las peticiones en la interfaz de su clase abstracta.
Busca reducir el acoplamiento entre módulos o sistemas (contexto no defina tantos métodos como clases existan, sino que tenga uno solo apuntando a la interfaz); relacionarse con abstracciones y no con concreciones.
Busca la transparencia con el cliente (los clientes no tienen que saber los tipos específicos de los objetos que usan; los clientes desconocen las clases que implementan dichos objetos).

- ***COI – Composite Reuse Principle*** (Strategy)
  En lugar de que los objetos se autoimplementen un comportamiento, pueden delegarlo a otros objetos. Usar composición nos permite reutilizar comportamiento (código) a través de la delegación (mecanismo de reúso a nivel de objetos), manteniendo un acoplamiento menor. Además, el cliente no conoce el interior de la clase contenida, respetando el encapsulamiento.

- ***EWV – Encapsulate What Varies*** (State – Strategy)
  Busca separar del código las partes propensas a cambiar y encapsularlas en otros objetos, para que cada vez que haya que hacer un cambio, el impacto esté localizado en una pequeña y única parte del sistema. Mejora la flexibilidad y mantenibilidad del sistema, y permite manejar la variabilidad

PRINCIPIOS SOLID

- ***SRP – Principio de responsabilidad única*** (Adapter – State – Iterator)
  Cada clase debería tener una única responsabilidad, un único motivo para cambiar.
  Intenta hacer a cada clase responsable de una única parte de la funcionalidad proporcionada por el software.

- ***OCP – Principio de abierto-cerrado*** (Adapter – Observer – State – Strategy – Iterator)
  Las clases (entidades de sw) tienen que estar cerradas para modificaciones y abiertas para extensiones de funcionalidad (agregar nuevas clases).
  La clase se crea y se cierra; si hay modificaciones en los requerimientos, se colocan nuevas clases con referencia a esa clase principal. No se colocan atributos o métodos a la clase ya creada, se incorporan a las nuevas clases.
  Ante la necesidad de un cambio, no debería cambiar el software existente, sino agregar algo nuevo que trabaje con el anterior sin modificar su funcionamiento.
  Otorga la máxima flexibilidad con el mínimo impacto.

- ***LSP – Principio de sustitución de Liskov*** (State – Template Method)
  (ayuda a predecir si una subclase es compatible con el código que funcionaba con objetos de la superclase).
  Evalúa si la estructura de la herencia está bien diseñada. Si el padre se ubica en el lugar del hijo y más o menos funciona, está bien diseñada/implementada. (Los subtipos deberían poder reemplazarse por sus tipos base sin romper el sistema; el hijo se debe comportar como el padre).

- ***ISP – Principio de segregación de interfaces*** (Adapter – Observer – Strategy)
  No forzar a los clientes a depender de métodos que no utilizan (interactuar o relacionarse con interfaces muy complejas, con cosas que no necesita), ni forzar a las clases del cliente (clases concretas) z implementar comportamientos que no necesitan.
  Se deben desintegrar las interfaces “gruesas” hasta crear otras más detalladas y específicas. Los clientes deben implementar únicamente aquellos métodos que necesitan de verdad.

❖ ***DIP – Principio de inversión de dependencia*** (Template Method)
No debemos hacer depender a clases de un nivel superior de clases de un nivel inferior.
Hay que desacoplar poniendo una interfaz en el medio (las clases de alto nivel van a depender de esa interfaz, en vez de depender de clases concretas de bajo nivel), entonces los módulos de bajo nivel proveen los servicios que requiere el módulo de alto nivel.

PATRONES DE DISEÑO

Son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software; es un concepto general para resolver un problema particular.

Un patrón es una solución concreta a un problema bien definido.

Son importantes porque ayudan a centralizar el conocimiento y porque ayudan a resolver un problema común de una manera mucho más eficiente, ya que no tenemos que pensar la solución a un determinado problema todas las veces que surja. Nos enseñan a resolver todo tipos de problemas utilizando principios de diseño.

¿A qué ayudan los patrones?

1) Encontrar objetos:
Ayudan a identificar abstracciones (clases como la estrategia o estados) no tan obvias, implementadas mediante clases de fabricación pura, y que son fundamentales para lograr un diseño flexible, bajar el acoplamiento y aumentar la cohesión.

2) Determinar la granularidad: (que tanto va a hacer una clase)
Cuánto comportamiento tienen las clases. Las clases de granularidad fina son las que implementan pocos métodos, mientras que las clases de granularidad gruesa son las que implementan muchos métodos.
El nivel de cohesión óptimo para las clases, es el que manejan las clases de granularidad fina, ya que, relacionándolo con el principio de responsabilidad única, buscamos hacer a cada clase responsable de una única parte de la funcionalidad proporcionada por el software, y que esa responsabilidad quede totalmente encapsulada en la clase.
Los patrones nos van a ayudar a decidir cuántos métodos implementamos en cada clase.

3) Especificar interfaces:
Interfaz → tipo especial de clase abstracta en la que todos sus métodos están vacíos.
Permiten ayudar a encontrar interfaces que van a contener el conjunto de peticiones que pueden realizar los clientes, y que ayudan a que estos no tengan que preocuparse por las distintas clases concretas que se van a encargar de esas peticiones.
Ayudan a *programar hacia la interfaz y no hacia la implementación*, lo cual favorece a que las soluciones que apliquemos sean transparentes y fáciles de entender para el cliente, aunque sean más difíciles programarlas.

4) Especificar implementación:
(hablamos de código que va dentro de los métodos. Hay ciertos patrones que ayudan a saber qué métodos debemos usar o qué algoritmos utilizar en los métodos).
La implementación de un objeto queda definida por su clase. La clase especifica los datos, la representación interna del objeto y define las operaciones que puede realizar.
Los patrones nos dan pautas para definir las clases participantes; nos ayudarán a entender la diferencia entre la herencia de clases y la herencia de interfaces.
La herencia de clases es un mecanismo para extender la funcionalidad de una aplicación reutilizando la funcionalidad de las clases padres; nos permite definir un nuevo tipo de objetos basándose en otros, heredando lo que necesita. Y la herencia de interfaces, describe cuándo un objeto se puede usar en el lugar del otro.

Teniendo en cuenta el PTI, manipular objetos en términos de la interfaz definida por las clases abstractas tiene dos ventajas: los clientes no tienen que conocer los tipos específicos de los objetos que usan; y los clientes desconocen las clases que implementan dichos objetos.
No se deben declarar las variables como instancias de clases concretas, sino que se ajustarán simplemente a la interfaz definida por una clase abstracta.

5) Favorecer la reutilización:
Pueden favorecer dos tipos de reúso; reúso de caja negra (los detalles internos de los objetos no son visibles) y reúso de caja blanca (las interioridades de las clases padres suelen hacerse visibles a las subclases)
Se trata de favorecer reúso de caja negra (composición) por sobre el reúso de caja blanca (herencia).
Optar por la composición de objetos frente a la herencia de clases ayuda a mantener cada clase encapsulada y centrada en una sola tarea.

6) Diseñar para el cambio:
Se debe diseñar el software de forma que sea flexible, robusto y fácil de evolucionar, ya que los requerimientos cambian. Los patrones nos ayudan a estar preparados para que hayan cambios en el software, y para que los podamos implementar al menor costo posible.

PATRONES DE CREACIÓN

Abstraen el proceso de creación y ocultan los detalles de cómo los objetos son creados, mejorando y flexibilizando tal proceso a través de la distribución de responsabilidades.

- **Singleton**: Garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella. (Nos permite acceder a un objeto desde cualquier parte del programa).
  Usar el patrón cuando en el programa necesitas tener una sola instancia disponible para todos los clientes (ej, un único objeto de base de datos compartido por diferentes partes del programa); o cuando necesitas un control estricto sobre las variables globales.
  - SRP: Puede cumplir con este principio si sólo se utiliza para una única responsabilidad, como administrar un recurso compartido, como lo puede ser la conexión con la base de datos.

PATRONES DE ESTRUCTURA

Determinan cómo combinar objetos y clases para definir estructuras complejas. Explican cómo ensamblar objetos y clases en estructuras más grandes.

- **Adapter**: Intermediario entre dos clases que no pueden comunicarse. Cuando existen clases incompatibles, adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
  - SRP: Puede separar la interfaz (o el código de conversión de datos) de la lógica comercial principal del programa.
  - OCP: Se pueden introducir nuevos tipos de adaptadores en el programa, sin romper el código del cliente (el que necesita la información) existente.
  - ISP: La interfaz intermedia tiene los comportamientos que le van a servir al cliente y los que tiene que implementar el adaptador.

PATRONES DE COMPORTAMIENTO

Manejan la comunicación entre objetos del sistema y el flujo de información entre ellos. Se encargan de delegar y distribuir las responsabilidades entre los objetos.

- **Observer**: Define una dependencia de uno a muchos entre objetos, de forma que cuando cambie el estado de un objeto, se notifica y se actualizan todos los objetos que dependen de él.

- OCP: Se pueden agregar nuevas clases suscriptoras sin tener que cambiar el código del publicador (y viceversa si hubiera una interfaz publicador).
- ICP: Tengo dos interfaces; pongo el comportamiento que tiene que ver con los sujetos en la ISujeto, para que los sujetos concretos lo implementen, y con el observador igual.
- PTI: Reducimos el acoplamiento entre sujetos y observadores porque se comunican a través de la interfaz.

- **State**: Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
  - SRP: Organiza el código relacionado a estados particulares en clases separadas.
  - OCP: Se pueden agregar nuevos estados sin cambiar las clases estado existentes o el contexto.
  - LSP: Los Concretos se pueden intercambiar con el Abstracto, ya que todos son capaces de responder a sus métodos.
  - DRY: El código común se mantiene en la implementación trivial del padre, evitando redefinirlo en todas las hijas.
  - EWV: Los concretos encapsulan la lógica específica de cada uno, aislando los cambios del resto, sin afectarlos.
  - TDA: El contexto delega el trabajo al Estado, que resolverá la lógica del pedido.

- **Strategy**: Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usa.
  - OCP: Se pueden agregar nuevas estrategias sin tener que cambiar el contexto.
  - ISP: Se relaciona una clase concreta con una interfaz que la implementa. La interfaz le da a las clases concretas los métodos que necesita para implementar.
  - DRY: La parte común del algoritmo se mantiene en el contexto, evitando duplicaciones.
  - EWV: Las partes del algoritmo que varían se encapsulan en las Estrategias.
  - PTI: Se establece una interfaz única y todas las estrategias se implementan hacia ella para utilizarse en el contexto.
  - COI: Se usa composición para que el contexto mantenga referencia a las estrategias.

- **Template Method**: Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
  - DIP: En pos de definir una estructura de algoritmo que se respete y no se cambie, puede ocurrir que el padre tenga una invocación concreta de una clase hija.
  - LSP: Analiza si está bien armada la herencia.

- **Iterator**: Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna. (Extraer el comportamiento de recorrido de una colección y colocarlo en un objeto independiente llamado Iterador).
  - SRP: Puede limpiar el código del cliente y las colecciones extrayendo algoritmos de recorrido voluminosos en clases separadas.
  - OCP: Puede implementar nuevos tipos de colecciones e iteradores, y pasarlos al código existente sin romper nada.
  - EWV: Un objeto iterador encapsula todos los detalles del recorrido. (Varios iteradores pueden recorrer la misma colección al mismo tiempo, independientemente unos de otros).
  - PTI: Todos los iteradores deben implementar la misma interfaz. (Código cliente sea compatible con cualquier tipo de colección o cualquier algoritmo de recorrido).
  - ISP: Al ofrecer una interfaz específica y limitada para la iteración.
  - DIP: Al depender de abstracciones en lugar de implementaciones concretas.
  - LSP: Al permitir que diferentes iteradores se comporten de forma intercambiable.

¿Qué se diseña?

- *Arquitectura*: Es el conjunto de decisiones significativas respecto a resolver requerimientos no funcionales y funcionales. Es diseño general, hay definiciones de cómo resolver requerimientos no funcionales frente a un contexto de infraestructura específico.
  Toma los requerimientos no funcionales significativos para la arquitectura y los aplica al modelo de análisis.
  Se representa a través de vistas, es un modelo genérico que no entra en detalles. La funcionalidad condiciona a la arquitectura; avanzan juntas, ya que se condicionan entre sí.
- *Procesos*: Transforma elementos estructurales de la arquitectura del programa en una descripción procedimental de los componentes del sw. Se utilizan patrones de diseño para la creación de procesos. A partir de las realizaciones de casos de uso de análisis se obtienen las realizaciones de casos de uso de diseño luego de considerar y aplicarles los RNF.
  Cómo vas a hacer para distribuir los componentes de sw en el hardware para cumplir con los requerimientos funcionales y no funcionales del sistema.
- *Datos*: Implica analizar nuestras clases y definir cuales de esas clases necesitan persistencia, en qué almacenamiento persistente las vamos a guardar, diseñar la BD y la estructura de los datos, y diseñar la persistencia (como hago para conectar las clases con la BD). Transforma los requerimientos en las estructuras de datos necesarias para hacer persistente el sw.
- *IHM*: Es una disciplina que se encarga de evaluar todos los aspectos que tienen que ver en cómo interactúan las personas con el sw. Esta disciplina combina las interfaces de usuario y la experiencia de usuario. Una tiene que ver con cómo ve el usuario el sistema, y la otra tiene que ver con cómo efectivamente funciona el sistema y cómo lo percibe el usuario.
- *Entrada/Salida*: Describe cómo se ingresa información al sw y cómo se presentarán las salidas del mismo.
  En lotes →se acumulan las transacciones para procesarlas después.
  En línea →en el momento en que ocurre la transacción, ésta es procesada.
  En tiempo real →son sistemas en línea, pero pueden modificar el ambiente donde está inmerso, tienen un tiempo estricto de respuesta.
- *Procedimientos manuales*: Describe cómo se integra el sw al sistema de negocio. Incluye los 'plan B' en caso de que el mismo falle. Nos permite insertar el sw en el negocio.

DISEÑO DE PERSISTENCIA

Persistencia → capacidad que tiene un objeto de permanecer en el tiempo y en el espacio.

DISEÑO DE PERSISTENCIA

Implica analizar nuestras clases, definir cuáles de esas clases necesitan persistencia, en qué almacenamiento persistente las vamos a guardar, diseñar la base de datos y la estructura de los datos, y diseñar la persistencia (cómo conectar las clases con la base de datos).

La persistencia la hemos hecho presente en la arquitectura como un componente "persistencia" que hacía de vínculo entre la lógica de negocio, que debía ser mantenido por un tiempo, y la base de datos relacional.

El origen de la problemática de necesitar del diseño de persistencia son los paradigmas, ya que el programa se diseña según el POO, y la base de datos trabaja con el paradigma relacional o estructurado.

- Hay que determinar qué objetos necesitan persistencia. Los objetos de entidad son los candidatos importantes para el almacenamiento persistente.
- (Es cierto que existen bases de datos de objetos, y utilizando este tipo de bases de datos no se necesita de ningún servicio de persistencia, pero no son las bases de datos más utilizadas; generalmente la mayoría de bases de datos que se utilizan en los programas son relacionales).
- Bases de datos relacionales: En este caso surgen problemas de incompatibilidad entre la representación de los datos orientada a registros y la orientada a objetos, por lo que se requiere un servicio especial para establecer la correspondencia.

- También podemos querer almacenar los datos en formato xml, en bases de datos jerárquicas, etc. De igual manera vamos a necesitar un servicio especial que haga que funcionen con objetos.

Problema de impedancia: Ocurre cuando queremos guardar objetos a BD relacionales. El primer problema es que las BDR no soportan comportamiento, sólo guardan datos; y el segundo problema es que no puedo guardar cualquier tipo de datos, porque sólo permiten primitivos
Se refiere al problema de la correspondencia entre los paradigmas. En el paradigma OO la información se almacena en los objetos, y las bases de datos relacionales pertenecen al paradigma estructurado. Por lo tanto, necesitamos transformar nuestra estructura de información de objeto a una estructura orientada a tablas. Esto crea un fuerte acoplamiento entre la aplicación y el DBMS.

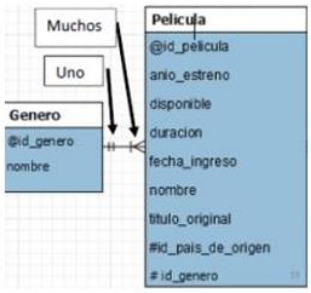
- Las bases de datos relacionales no soportan comportamiento, los métodos no son soportados, todos los métodos que hacen la integración de la clase y que es lo más importante que tiene la clase no se puede guardar en una base de datos relacional.
- Las bases de datos relacionales sólo almacenan tipos de datos primitivos y no objetos complejos, estos tipos de datos los define el proveedor del motor de base de datos, y no se pueden definir tipos propios de datos y las clases no son ni más ni menos que un tipo de datos propio. Hay que desarmar la estructura de clases y mapearlas a cada una de las columnas de las tablas.

La vinculación entre el modelo de clases y la base de datos relacional es el MAPEO o diseño de persistencia.

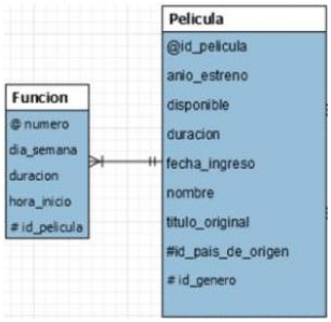
Las clases persistentes se les asigna una tabla y se les descarta el comportamiento. A cada atributo se lo coloca en una columna de la tabla. La tabla se llama igual que la clase para mantener la trazabilidad o el vínculo. Cada atributo primitivo se transformará en una columna en la tabla. Si el atributo es complejo agregamos una tabla adicional o distribuimos el atributo en varias columnas de la tabla de la clase. La columna de la clave primaria será el identificador único de la instancia. El identificador debe ser preferentemente invisible al usuario y generadas automáticamente por máquina.

- Opciones para el mapeo de asociación

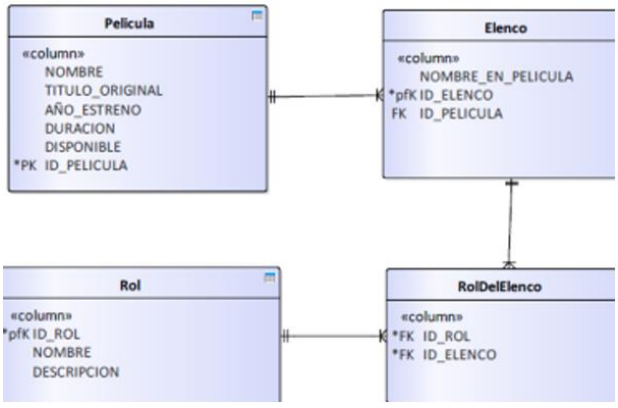
1. 1 a muchos



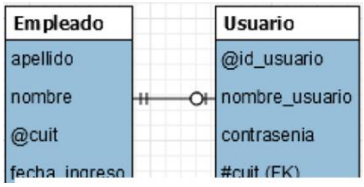
2. Muchos a uno



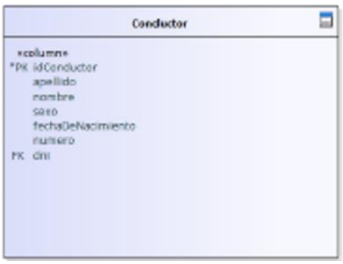
3. *Muchos a muchos*: se crea una tabla intermedia asociativa. Es necesaria por el problema de la clave foránea.

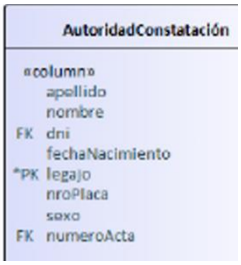


4. *Uno a uno*: las relaciones uno a uno de ambos lados no debería existir porque en ese caso debería fusionar todo en una sola tabla. En este caso si se puede porque del lado del usuario tenemos opcionalidad 0..1.

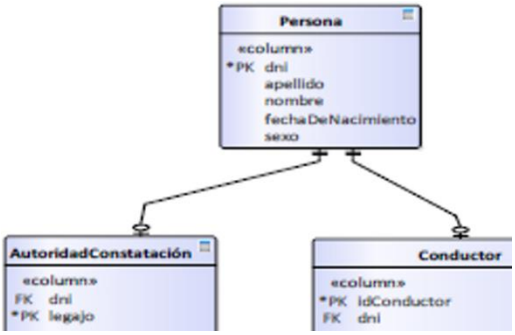


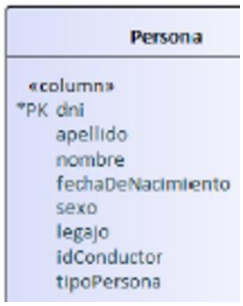
- Opciones para el mapeo de la herencia
  1) *Eliminar al padre*: significa eliminar la herencia. Quedan tablas únicas para los objetos hijos. Se copian todos los atributos de la clase padre. Ninguna tabla representa la clase abstracta. Es más rápida ya que los datos están en una sola tabla y no hay que hacer joins. El tamaño de la tabla aumenta ya que se duplican columnas (los atributos comunes van a repetirse). Puede traer problemas de consistencia y redundancia.





  2) *Simularla*: es una jerarquía supertipo-subtipo. La clase abstracta está en su propia tabla y las tablas de los hijos hacen referencia a ella. Reduce la redundancia, pero hacer JOINs que pueden reducir la performance.





3) *Eliminar los hijos*: se ponen todos los atributos de todas las clases hijas en una sola tabla. No se recomienda porque no hay trazabilidad, hay baja cohesión y se desperdicia almacenamiento ya que muchos atributos quedarán en blanco.

Para elegir entre las distintas opciones tengo que analizar el volumen de las tablas, la cantidad de registros y cómo va a ir creciendo el modelo.

¿a qué clase de nuestro software le vamos a dar la responsabilidad de que haga el trabajo de hacer las clases persistentes? → por esto se utiliza un framework de persistencia.

Framework de persistencia

Es un conjunto de tipos de propósito general, reutilizable y extensible que proporciona funcionalidad para dar soporte a los objetos persistentes.

Un servicio de persistencia o subsistema realmente proporciona el servicio, y se creará con un framework de persistencia.

Un framework de persistencia contiene clases concretas y abstractas que definen interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras variantes. Ofrece un alto grado de reutilización. Debería proporcionar funciones para almacenar y recuperar objetos en un mecanismo de almacenamiento persistente, y confirmar y deshacer las transacciones.

Generalmente, un servicio de persistencia tiene que traducir los objetos a registros y guardarlos en una base de datos, y traducir registros a objetos cuando los recuperamos de la base de datos.

- Materialización: Una o varias filas de tablas de la base de datos son convertidas a objetos en memoria.
  - Materialización inmediata: recupera los objetos al instante.
  - Materialización perezosa: recupera los objetos cuando se los necesita realmente.

- Desmaterialización: Proceso inverso; un objeto se graba en la base de datos en forma de uno o varios registros de tablas.

Los modelos de persistencia que tenemos son:

Superobjeto persistente

Se propone crear una clase de fabricación pura (SuperObjetoPersistente), y en esa clase definir el método de materialización y desmaterialización, junto con cualquier método que necesitemos para resolver la persistencia.

Entonces, todas las clases que necesiran persistencia deberían heredar de la clase SuperObjetoPersistente.

Se llama correspondencia directa porque no hay nada en el medio entre la clase de fabricación pura y las clases que necesitan persistencia. El SuperObjetoPersistente define los comportamientos, además de tener la responsabilidad de saber cómo hacerse persistentes en una base de datos.

Ventajas:

- ✓ No crece la estructura de clases.
- ✓ Fácil de implementar.

Desventajas:

- ✓ Si la clase trae herencia del dominio y también tiene que heredar la persistencia, hay muchos lenguajes de programación que no soportan la herencia múltiple, sólo herencia simple. Aunque se podría solucionar con una realización de interfaz para el SuperObjetoPersistente.
- ✓ No se cumple e principio de diseño open-closed, ni tampoco el principio de responsabilidad única. Una clase de dominio no debería saber cómo hacerse persistente porque no modela ese concepto.
- ✓ Se rompe la cohesión.
- ✓ Desde el punto de vista arquitectónico, se presenta el problema de acoplamiento entre la capa de negocio y la capa de base de datos.

Esquema de persistencia

Se basa en un esquema de correspondencia indirecta. Es lo que se usa desde la arquitectura y es lo que se recomienda. Para que funcione, se agrega en el esquema de persistencia una clase que funcione como intermediaria para clase de negocio. Esas clases tienen el método de materializar y desmaterializar implementado. Además, contienen la PK y el puntero, y se puede mantener la relación entre ambos paradigmas. Le quitamos la responsabilidad a las clases de negocio de saber cómo materializarse y desmaterializarse.

Es una decisión arquitectónica. Agrego una capa (intermedia) de persistencia entre la capa Lógica de Negocios y la BD, compuesto por clases de fabricación pura, que tienen el rol de establecer vínculos entre la capa de Lógica de Negocio y la capa de Datos, sin que los objetos se vean afectados por el problema de persistencia. El esquema de persistencia resuelve el problema de materializar, desmaterializar y los métodos necesarios para la persistencia.

El esquema de persistencia tiene un conjunto de clases o interfaces que definen los métodos, y para integrar este esquema a mi modelo de dominio, tengo que crear una clase de fabricación pura por cada clase del dominio que necesite persistencia.

Ventajas:

- ✓ Se respeta el principio de responsabilidad única y open-closed porque las clases de dominio no se tocan.
- ✓ Mantiene alta la cohesión.
- ✓ Soporta extensibilidad.
- ✓ Mejora la reusabilidad.
- ✓ Mantiene bajo el acoplamiento, ya que aplica el principio de inversión de dependencias. El control lo toman las clases del esquema de persistencia, que van y buscan a los objetos, y los materializan y desmaterializan.

Desventajas:

- ✓ Hay un aumento considerable de clases.

Patrones de persistencia

- *Patrón identificador de objetos*: Propone asignar un identificador de objeto a cada registro y objeto.
- *Patrón de representación de objetos como tablas:* Propone la definición de una tabla en una BDR por cada objeto persistente.
- *Patrón fachada:* Permite el acceso al servicio de persistencia mediante fachada, proporcionando una interfaz uniforme a un subsistema. La fachada no hace el trabajo, sólo lo delega en objetos del subsistema.
- *Patrón conversor o broker:* Propone crear una clase responsable de materializar y desmaterializar los objetos. Agrega un intermediario por cada clase de dominio.
- *Materialización con el método plantilla:* Con el patrón template method encapsulamos en una sola clase el comportamiento común de la materialización. El punto de variación es la manera de crear el objeto a partir del almacenamiento.
- *Patrón de gestión de caché:* Es conveniente mantener los objetos materializados almacenados en una caché local para mejorar el rendimiento y dar soporte a las operaciones de gestión de transacciones, con un OID como clave. El conversor primero busca en la caché para evitar materializaciones innecesarias.
- *Estados transaccionales y el patrón state:* Los objetos persistentes pueden insertarse, eliminarse o modificarse. Operar sobre un objeto persistente no provoca una actualización inmediata en la base de datos, más bien se debe ejecutar una operación commit explícita.
- *Diseño de transacción con patrón command:* Con el patrón se busca crear un objeto por cada operación a realizar en la transacción, haciendo una cola de operaciones permitiendo hacer un rollback, si es necesario.
- *Materialización perezosa mediante proxy virtual:* Se basa en la suposición de que los proxies conocen el OID de sus sujetos reales, y al requerir la materialización, se lo usa para identificar y recuperar el sujeto real.

UNIDAD 3

ARQUITECTURA

Conjunto de decisiones significativas que tomamos respecto de cómo resolver los requerimientos no funcionales (o de calidad), teniendo en cuenta el contexto (reglas de negocio) donde va a funcionar el sistema.

Los sistemas se descomponen en subsistemas. El proceso de diseño para identificar estos subsistemas y establecer un marco de trabajo para control y comunicación de los subsistemas se llama “diseño arquitectónico”.

DISEÑO ARQUITECTÓNICO

Diseño estratégico que se encarga de la asignación de modelos de requerimiento esenciales a una tecnología específica. Proceso que toma el modelo de análisis y los requerimientos no funcionales que no tuvimos en cuenta antes, y los aplica sobre una tecnología específica; planifica a un nivel alto de abstracción, lo que va a ser el diseño y la implementación del software.

Se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema.

Es el “plan de diseño” → donde se toman las decisiones de cómo vamos a resolver esos requerimientos.

La arquitectura captura la estructura del sistema en términos de componentes y cómo éstos interactúan.