Práctica 4: CWS2

Objetivos

- Separar el código de la capa de negocio en dos capas: lógica de negocio y persistencia.

Material necesario

- La solución a la práctica 3 que se encontrará en el proyecto CWS1

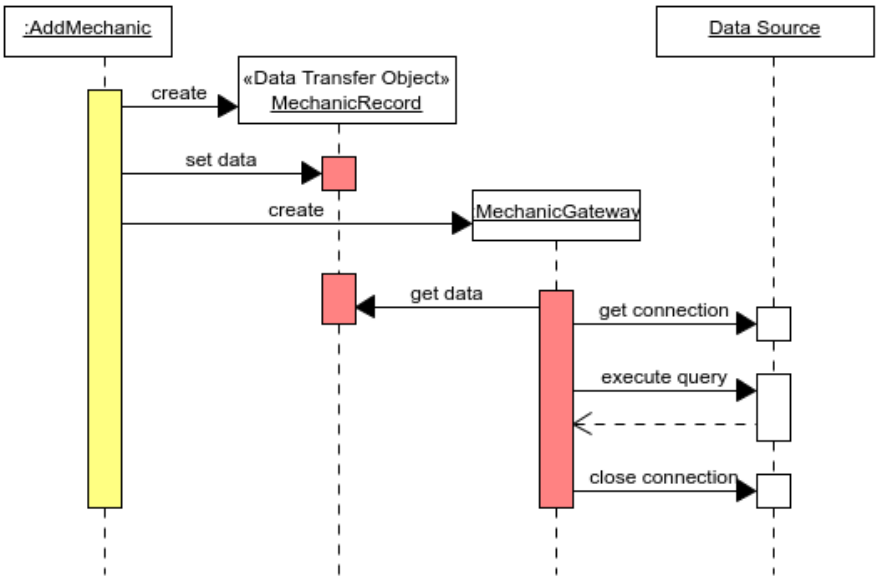
Ejercicio 1: Separar lógica de negocio y acceso a datos

Se modifica el diseño para independizar las clases de negocio de las de persistencia. Para lograrlo se aplicarán los patrones Layering, Transaction Script (TS) y Table Data Gateway (TDG).
Patrón Layering: Cree un paquete **cws.application.persistence** para contener el código de acceso a datos (DAL o Data Access Layer), eliminándolo de la capa business (BL o Business Layer).

Ejercicio 2: Organizar el código de la capa de acceso a datos aplicando el patrón Table Data Gateway y el código de la capa de negocio utilizando el patrón Transaction Script

Patrón TDG

Usando el patrón TDG, los objetos de negocio acceden a los datos a través de **objetos Gateway**. Cada Table Data Gateway contiene **todo el código (incluido el SQL) para acceder a una sola tabla**: busca, inserta, actualiza y elimina. Los objetos de negocio invocan métodos en los objetos del Gateway para almacenar y recuperar datos de la base de datos.



La **estructura de paquetes** será semejante a la de BL

* Paquete **persistence** incluye todo el código de acceso a datos.
* Un subpaquete para cada tabla (persistence.mechanic, persistence.invoice,...) que contendrán el código que se encarga de las operaciones sobre esa tabla concreta.

Norma de estilo: Como norma general, los nombres de los subpaquetes será el mismo que las tablas en la base de datos, en singular y sin la T inicial.

- El **paquete persistence** contendrá una interfaz **Gateway.java** (ver código adicional → persistence → Gateway interfaces) que define operaciones comunes sobre cualquier tabla (insertar, borrar, recuperar o actualizar).
- Se pueden añadir métodos de **recuperación** de datos para alguna tabla. En los **subpaquetes** persistence.XXX (persistence.mechanic, persistence.invoice, ...) habrá interfaces que extienden la anterior (ver código adicional → persistence → Gateway interfaces). Por ejemplo, **MechanicGateway** extiende Gateway añadiendo el método **findByDni**.
- Cada subpaquete (persistence.**) contendrá un subpaquete **impl** (persistence.mechanic.impl) que contendrá la clase que implementa el gateway.

Norma de estilo: El nombre de esas clases será el mismo que las interfaces java con el sufijo **Impl.**

- Los Gateways recuperarán datos de las tablas en forma ResultSet pero los intercambiarán con la capa de negocio como objetos java (nuevos dto, por ejemplo MechanicDALDto).
  * Vienen dentro de los gateway.
  * Los DTOs solamente encapsulan datos, no incluyen ninguna lógica de negocio.

Norma de estilo: Como norma general, los nombres de los DTOs entre la capa de persistencia y la de negocio y viceversa serán <nombre de la tabla en singular y sin la T inicial><DALDto> (MechanicDALDto, InvoiceDALDto, ...). Los nombres de los campos serán exactamente los nombres de los campos de las tablas, todo en minúscula.

- Se proporcionan clases **Assembler** (MechanicAssembler), cuya tarea fundamental será transformar datos crudos de la BBDD en DTO's.

Antes de implementar la interfaz MechanicGateway y eliminar el código de acceso a datos de los objetos de negocio utilizando en su lugar los Gateway para recuperar/actualizar la información de las tablas, es imprescindible conocer el siguiente patrón.

Patrón Transaction Script

La lógica de negocio se organiza como procedimientos; uno por cada acción solicitada por el usuario (borrar mecánico, facturar trabajos, etc). Estos procedimientos se llaman Transaction Script (AddMechanic, DeleteMechanic, ... serán TS).

Cada **TS** genera **una sola sesión transaccional de comunicación con la base de datos**:

- La conexión con la BBDD se realiza en el Transaction Script.
- Se debe garantizar que los Gateway invocados desde un TS utilicen, **TODOS ellos**, esa misma conexión, para garantizar una única transacción. Se debe desactivar el modo autocommit.

- La operación en el TS debe ser transaccional (commit, rollback)

La estructura general de un TS, sería:

- Crear los gateways que se necesiten .
    * AddMechanic necesita MechanicGateway
- Obtener conexión con la base de datos que será utilizada en los gateways (ver **util.jdbc.createThreadConnection**, **util.jdbc.getCurrentConnection**).
- Desactivar el modo autocommit.
- Realizar las operaciones pertinentes utilizando, si fuese necesario, el/los gateways para recuperar la información o actualizarla.
- Confirmar la transacción.
- Cerrar la conexión.
- Retornar resultado (si lo hubiera) a la capa de presentación (PL). Los objetos de transferencia de datos entre PL y BL siguen siendo los mismos (MechanicDto, InvoiceDto, ...).
- Liberar recursos
- En caso de que se produzca alguna excepción durante la ejecución, es necesario realizar rollback.

El código de los objetos de negocio puede verse modificado sensiblemente al dividirlo entre las dos capas, BL y DAL. ¡ Cuidado!

Para intercambiar información con el Gateway, la clase Assembler tiene métodos (bajo el comentario para Sesión 4) que son capaces de construir DTO de la capa de negocio a partir de Dto de la capa de persistencia.
También se proporcionan Assembler para la capa de persistencia (incluídos en el Gateway correspondiente) que son capaces de transformar un ResultSet en un Dto de la capa de persistencia.

Patrón Simple Factory

Por último, aplicar el patrón Simple Factory en la capa de persistencia, igual que se hizo en la capa de negocio.

- Se proporciona la factoría, **PersistenceFactory** y se requiere su uso en la capa de negocio para evitar referencias a objetos de implementación de la DAL.
- En este caso, serán necesarias más interfaces y DTOs para evitar errores de compilación (incorporar las que se proporcionen en la carpeta de código adicional o comentar las líneas necesarias).

Curso de acción recomendado para un caso concreto: añadir mecánico.

1. Asegúrese de que las siguientes clases se encuentran en su proyecto. De lo contrario, añádalas/impleméntelas:
    a. La clase MechanicDALDto está incluida. Esta clase sirve para comunicar las capas de negocio y persistencia y será muy similar a la MechanicBLDTO utilizada en la sesión anterior.
    b. La interfaz Gateway está incluida.
    c. Hay una interfaz MechanicGateway que extiende Gateway.
    d. Hay una clase MechanicGatewayImpl que implementa los métodos de MechanicGateway.
2. Este caso de uso precisa buscar mecánicos por DNI para asegurarse de que no existe ya un mecánico en la base de datos con el DNI del mecánico que intentamos introducir en la BBDD. Para hacer esta comprobación:
    a. La clase MechanicGatewayImpl ha de proporcionar una implementación para el método findByDni. En esencia, el código para hacer esta operación ya fue desarrollado en la sesión anterior y se debería encontrar en FindMechanicByDni (ejecución de la consulta, transformación en DTO…). No obstante, MechanicGatewayImpl pertenece a la capa de persistencia, por lo que este método debe retornar MechanicDALDto en lugar de MechanicBLDto.
    b. Ahora, el execute() del comando FindMechanicByDni consistirá en una llamada al método findByDni de un objeto MechanicGatewayImpl. El retorno de esta llamada será un MechanicDALDto que debe ser transformado aquí en un MechanicBLDto. Usa la clase MechanicAssembler para realizar esta transformación.
3. Por supuesto, este caso de uso también requiere que el mecánico quede finalmente añadido a la base de datos. Para ello proporciona una implementación al método add() de la clase MechanicGatewayImpl. Repita los pasos de refactorización realizados en el punto anterior para mover código del comando AddMechanic a MechanciGatewayImpl.
4. En este punto, la aplicación es capaz de añadir mecánicos y, además, las capas de negocio y persistencia están bastante desacopladas. No obstante, el comando AddMechanic probablemente esté ejecutando un new MechanicGatewayImpl() para crear un objeto MechanicGateway con el que interaccionar con la base de datos. Podemos desacoplar aún más estas dos capas si utilizamos una factoría:
    a. Compruebe que su proyecto incluye la clase **PersistenceFactory** (se proporciona en el código adicional).
    b. Llame a **PersistenceFactory.forMechanic()** desde AddMechanic para obtener un objeto MechanicGatewayImpl. Además, asegúrate de que AddMechanic usa una variable de tipo **MechanicGateway** para almacenar este objeto (no MechanicGatewayImpl).

IMPORTANTE: como quizá se haya dado cuenta, en este punto el patrón Transaction Script no está completamente implementado. Todas estas operaciones deberían ocurrir en el transcurso de una única transacción, sin embargo, hay dos llamadas a un Gateway (findMechanicById y

addMechanic) que están utilizando dos conexiones independientes a la base de datos. Este problema será solucionado durante el ejercicio 5 de este mismo guion.

Ejercicio 3: Cambiar SQLException a PersistenceException

La excepción SQLException está muy relacionada con ciertos tipos de bases de datos y, para desacoplar el negocio del sistema gestor de bases de datos subyacente, se debe crear un nuevo tipo de excepción llamada **PersistenceException**.

La capa de acceso a datos convertirá SQLException en PersistenceException y la capa de negocio recogerá la excepción PersistenceException, que pudo haber sido lanzada por cualquier base de datos subyacente.

Ejercicio 4: Extraer del código las líneas SQL de acceso a datos

Crear (o reutilizar) un fichero de propiedades (configuration.properties) al que se copiarán todas las líneas SQL que se encuentren en el código.

- Utilice la clase Conf (ver código adicional) para leer del fichero de propiedades las cadenas de texto con las sentencias SQL.
- Organice el fichero de propiedades de la siguiente forma:
    * Todas las sentencias sql que se refieran a la misma tabla irán juntas, precedidas por una línea de comentario que indique el nombre de la tabla.
    * Organiza las sentencias alfabéticamente (TCLIENTS_XXX, irá antes que TVEHICLES_XXX).
    * Las propiedades seguirán el siguiente patrón de nombres, todo en mayúsculas y separados por un guion bajo:
      Nombre de la tabla_Nombre del método del Gateway que usa la sentencia

Ejercicio 5: Completar el código

Revise cuidadosamente la interfaz del servicio y compruebe que las implementaciones se corresponden con su javadoc, punto por punto. Es posible que obligue a implementar nuevos objetos de negocio. Por ejemplo:

- Para añadir o actualizar un mecánico, el dto y todos los campos del dto deben ser no vacíos. En caso contrario, se lanzará IllegalArgumentException.
- Cuando se añada un mecánico, no debe existir otro con el mismo DNI, mientras que cuando se actualice debe existir. En caso contrario, se lanzará BusinessException.
- Para borrar, el id recibido debe ser no vacío. En caso contrario se lanzará IllegalArgumentException.

Ejercicio 6: Aplicar el patrón Command en la Business Layer

La organización de la capa de negocio sigue el patrón Transaction Script. Este patrón organiza toda la lógica de negocio como procedimientos (procesos de negocio) que implementan lógica de

una determinada acción solicitada por la interfaz de usuario y acceden a la base de datos generando una sola sesión transaccional de comunicación con ella.
Este patrón puede ser utilizado de forma más elegante, si lo ayudamos con el patrón **Command**.
Algunos de los roles de este patrón se proporcionan ya implementados:

- **Command interface**: Se proporciona en additional classes → business → util
- **Invoker**: Se proporciona una clase java (CommandExecutor) en la misma carpeta. Se ocupa de iniciar la ejecución de los comandos concretos. Agrupa todas las acciones comunes a todos los comandos.
  Sin embargo, será necesario ajustar nuestras clases para adaptarse a otros:
- **Cliente**: La clase MechanicCrudServiceImpl crea y configura objetos de tipo command. En el constructor, le pasa a cada uno de ellos los parámetros con los que realizar la acción.
- **Concrete Command** y **Receiver**: Juntas en nuestra implementación. Deben modificarse extrayendo de ellas el código que se repite, que se realizará en el CommandExecutor.

Ejercicio 7: Repetir los patrones anteriores con el caso generar factura

Como **trabajo autónomo**, deberá repetir lo anterior con el código de **facturar trabajos** y con el de **buscar trabajos no facturados por dni**.