Práctica 4: CWS2

Objetivos

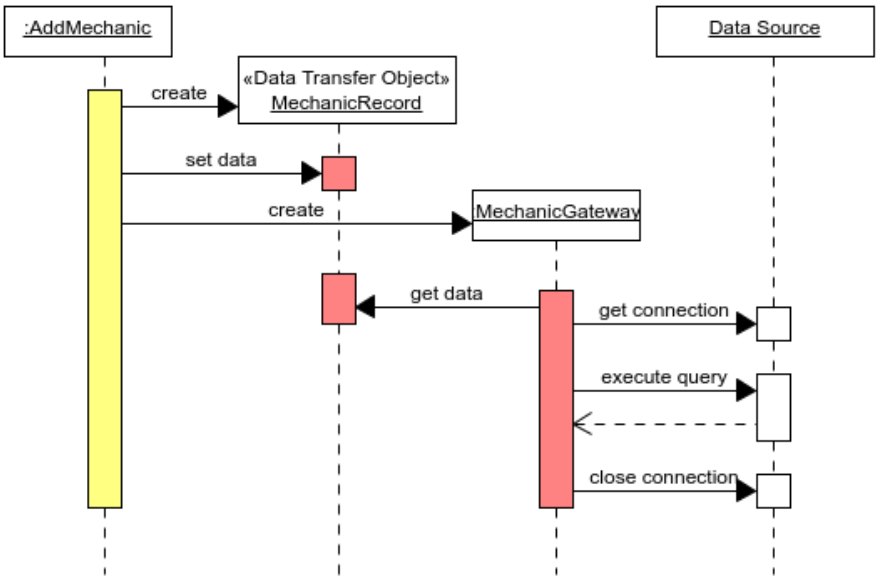
- Separar el código de la capa de negocio en dos capas: lógica de negocio y persistencia.
- La solución a la práctica 3 que se encontrará en el proyecto CWS1

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 Layers, 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 service (BL o Service Layer).

Ejercicio 2: Organizar el código de la capa de acceso a datos aplicando el patrón Table Data Gateway

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 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 SL

* 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) que define operaciones comunes sobre cualquier tabla (insertar, borrar, recuperar o actualizar) y lanza PersistenceException (disponible en código extra) en lugar de SQLException.
- En los **subpaquetes** persistence.*** (persistence.mechanic, persistence.invoice, …) habrá interfaces que extienden la anterior. Por ejemplo, **MechanicGateway** (código extra) puede extender Gateway añadiendo el método **findByNif**. Estos gateways más específicos se pueden modificar **añadiendo métodos únicamente de recuperación de datos**.
- Cada subpaquete (persistence.*) contendrá un subpaquete **impl** (persistence.mechanic.impl) que contendrá las clase que implementan el gateway.

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

- Los Gateways recuperarán datos de las tablas en forma ResultSet pero los intercambiarán con la capa de negocio como objetos de transferencia de datos o DTO. Para no confundir estos nuevos dto con los que transfieren información entre la capa servicio y la interfaz con el usuario, nombraremos a estos nuevos dto con el sufijo **Record**, por ejemplo MechanicRecord.
    * Se declaran dentro de los gateway.
    * Sólo 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><Record> (MechanicRecord, InvoiceRecord, …). Los nombres de los campos serán exactamente los nombres de los campos de las tablas, todo en minúscula.

- Se proporcionan clases **RecordAssembler**, cuya tarea fundamental será transformar datos crudos de la BBDD en Records como los mencionados aquí.
- Al implementar la interfaz MechanicGateway (y otras) se elimina 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.
- **El código de los objetos de negocio puede verse modificado sensiblemente al dividirlo entre las dos capas, servicio y acceso a datos. ¡Cuidado!**

Ejercicio 2.1: Extraer del código las líneas SQL de acceso a datos

Crear (o reutilizar) dos ficheros de propiedades a los que se copiarán todas las propiedades de conexión (configuration.properties) y sentencias SQL (queries.properties) que se encuentren en el código. De ambos ficheros se proporcionan esqueletos iniciales en el código extra (src) que contienen las sentencias SQL que ya venían en el código inicial..

- Fichero configuration.properties contendrá URL, USERNAME y PASSWORD. Utilice la clase Conf proporcionada para leer del fichero de propiedades las cadenas de texto.
- Fichero queries.properties contendrá las sentencias SQL. Utilice la clase Queries proporcionada para leer las sentencias de este fichero. Organice el fichero como se indica:
    * 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_***, irá antes que TVEHICLES_***).

* 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 2.2: 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 y la transforma en RTE.

Ejercicio 2.3: Patrón Simple Factory

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

Se proporciona la clase PersistenceFactory para su uso en la capa de negocio y así evitar referencias a objetos de implementación de la capa de acceso a datos.

También se proporciona una nueva versión de la clase ServiceFactory, que debe sustituir a la anterior.

Ejercicio 2.4: Factories

En el código extra se proporciona una clase **Factories** (uo.ri.conf) a través de la que se accede a cada una de las factorías anteriores, haciendo innecesario crear una instancia de cada una de ellas cada vez que se necesiten.

Ejercicio 3: Organizar el código de la capa de servicio utilizando el patrón Transaction Script

La lógica de negocio se organiza como procedimientos, uno por cada acción solicitada por algún cliente (borrar mecánico, generar factura, añadir proveedor, generar pedido, etc). Estos procedimientos se llaman Transaction Script. AddMechanic, DeleteMechanic, … serán TS.
Para intercambiar información con el Gateway, las clases DtoAssembler de la capa de servicio tienen métodos que son capaces de construir DTO de la capa de servicio a partir de Dto de la capa de persistencia (Record). Los DtoAssembler iniciales están en el código extra.

Ejercicio 3.1: TS transaccionales.

Cada **TS** debe ejecutarse en **una sola sesión transaccional de comunicación con la base de datos**. Se debe garantizar que los Gateway invocados desde un TS utilicen, **TODOS ellos**, la misma conexión, para garantizar una única transacción. Sin embargo, cada operación invocada en un

Gateway crea una conexión. **La gestión de la conexión y la transacción debe hacerse en los objetos de negocio, de ahí su nombre transaction script**.

La estructura general de cualquier TS, sería:

- Crear los gateways necesarios. Por ejemplo, AddMechanic necesita MechanicGateway
- Obtener conexión con la base de datos. Esto implicará modificar los gateways. Se proporciona una clase adicional Jdbc (uo.ri.cws.application.persistence.util.jdbc) que se ocupa de crear una conexión y guardarla en el hilo de ejecución para recuperarla más tarde en cada gateway.
- Desactivar el modo autocommit.
- Realizar las operaciones pertinentes utilizando, si fuese necesario, el/los gateways para recuperar la información o actualizarla.
- Confirmar la transacción.
- Retornar resultado (si lo hubiera) a la capa de presentación. Los objetos de transferencia de datos entre UI y service siguen siendo los mismos (MechanicDto, InvoiceDto, …).
- Liberar la conexión.
- En caso de que se produzca alguna excepción durante la ejecución, es necesario realizar rollback. Si la excepción es SQLException, transformar en RuntimeException.

Ejercicio 4: Aplicar el patrón Command en la Service Layer

La organización de la capa de negocio sigue el patrón Transaction Script. 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 la interfaz Command (service.util.command). Los TS deben implementarla.
- **Invoker**: Se proporciona la clase JdbcCommandExecutor (service.util.command.executor). 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. Ejecuta el comando utilizando el JdbcCommandExecutor.
- **Concrete Command** y **Receiver**: Juntas en nuestra implementación. Deben modificarse extrayendo de ellas el código que se repite, que se realizará en el JdbcCommandExecutor.

Ejercicio 5: 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 nif**.