Programming exercise 4: CWS2

Goal

- Split business layer code in two layers: business and persistence

Getting started

- You need the project solved in the previous session (CWS1).

Exercise 1: Separate business layer from Data Access layer

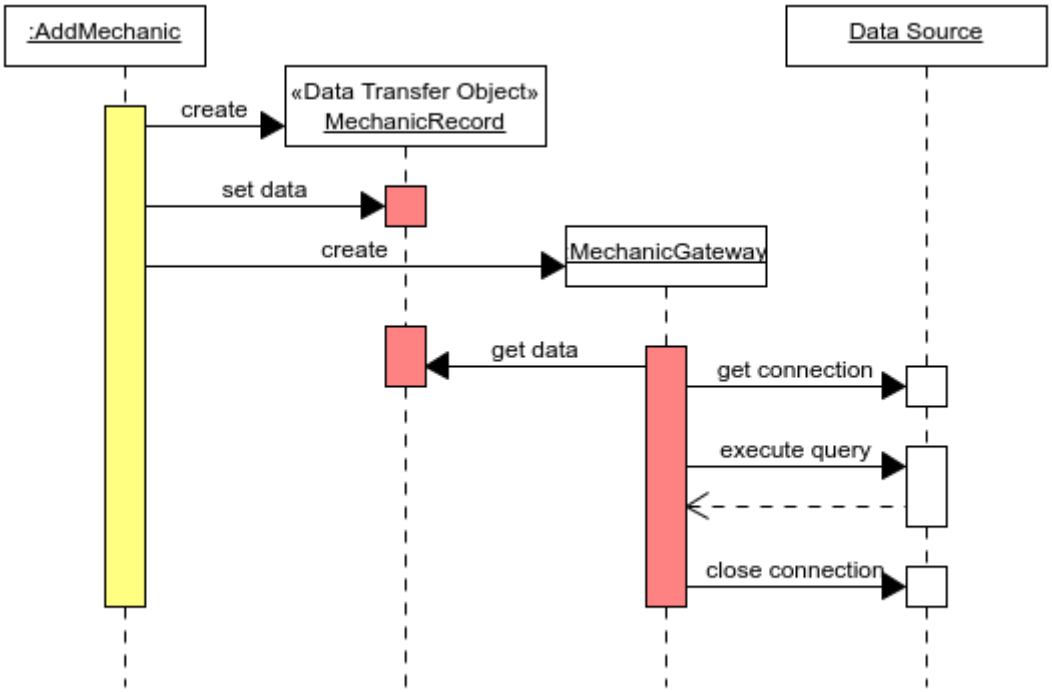
The application is redesigned to make business layer independent of persistence layer. To achieve this, the Layering, Transaction Script (TS) and Table Data Gateway (TDG) patterns will be applied.

Layering Pattern: Create a **cws.application.persistence** package to contain the data access code (DAL or Data Access Layer), removing it from the business layer (BL or Business Layer).

Exercise 2: Use Table Data Gateway and Transaction Script patterns to refactor business and data access layers.

TDG pattern

Using TDG pattern, **business objects** will access **data source** through **Gateway objects**. A simple Table Data Gateway holds **all the SQL for accessing a single table:** finds, insert, update and delete. Business objects invoke methods in the **table data gateway objects** to store and retrieve data from database. The latter will pass back a suitable data (**data transfer object**).



- **PACKAGE STRUCTURE**
    * New package **persistence** to include data access layer classes.
    * One **sub-package for each table** (persistence.mechanic, persistence.invoice…). Classes in a subpackage will be responsible for encapsulating access to one database table.

Style guide: Generally, the names of the subpackages will be the same as those of the tables in the database, in singular and without the initial **T**.

- Package **persistence** will contain interface **Gateway.java** (see extra code → persistence → Gateway interfaces). This java interface defines **access methods common to every gateway** (create, read by identifier, read all, update and delete).
- It is possible to **extend Gateway adding more methods** but **only retrieving** data methods. So, there could be more interfaces in the **subpackages** (persistence.***), **extending Gateway** (see extra code). For example, **MechanicGateway** will extend Gateway by adding a new method **findByNif**.
- Each package (persistence.*) will contain a subpackage **impl** (persistence.mechanic.impl). Classes implementing the corresponding Gateway interfaces will be here.

Style guide: Names of these implementation classes will be the same as those of the java interface with the **Impl** suffix.

- Gateways will retrieve data from tables in ResultSet form but exchange it with the business layer as data transfer objects or DTOs. In order not to confuse these new dtos with those that transfer information between the service layer and the user interface, we will name these new dtos with the suffix **Record**, for example MechanicRecord.
    * They should be inside the gateways interfaces.
    * They do not include any logic; they only encapsulate data.

Style guide: Generally, the names of these DTOs (carrier between persistence and business layers) will be *<table name in singular and removing initial T><Record>* (MechanicRecord, InvoiceRecord …). The names of the fields will be exactly as the names of the fields in the tables, all the letters in the field name lowercase.

- RecordAssembler **classes are provided**, whose main task will be to transform raw database data into Records such as those mentioned here.
- Implementing the MechanicGateway interface (and others) removes the data access code from business objects by using the Gateways instead to retrieve/update information from tables.
- **The code of business objects can be significantly modified by dividing it between the two layers, service and data access. Care!**

Exercise 2.1: Extract SQL lines of data access from code

Create (or reuse) two property files to which all connection properties and SQL statements found in the code will be copied.

- File configuration.properties will contain URL, USERNAME and PASSWORD. Use the provided Conf class to read text strings from the property file.
- The queries.properties file will contain the SQL statements. Use the Queries class provided to read the statements in this file. Organize the file as instructed:
    * All sql statements that refer to the same table will be together, preceded by a comment line indicating the name of the table.

* Organize sentences alphabetically (TCLIENTS_, it will come before TVEHICLES_).
* The properties will follow the following naming pattern, all capitalized and separated by an underscore:

  Name of the table_Name of the gateway method that uses the statement

Exercise 2.2: Changing SQLException to PersistenceException

The SQLException exception is closely related to certain types of databases, and to decouple the business from the underlying database management system, a new exception type called **PersistenceException must be created**.

The data access layer will convert SQLException to PersistenceException, and the business layer will pick up the PersistenceException, which could have been thrown by any underlying database, and transform it into RTE.

Exercise 2.3: Simple Factory Pattern

Apply the Simple Factory pattern on the persistence layer, just as you did on the business layer.

The PersistenceFactory **class is provided** for use in the business layer to avoid references to data access layer deployment objects.

Exercise 2.4: Factories

The extra code provides a Factories class (uo.ri.conf) that instantiates each of the above factories, making it unnecessary to instantiate each of them every time they are needed.

Exercise 3: Organize Service Layer Code Using the Transaction Script Pattern

Business logic is organized as procedures; one for each action requested by the user (delete mechanics, invoice jobs, etc.). These procedures are called Transaction Script (AddMechanic, DeleteMechanic, ... will be TS).

To exchange information with the Gateway, the DtoAssembler classes of the service layer have methods that can construct the DTO of the service layer from the Dto of the Persistence Layer (Record). The initial DtoAssemblers are in the extra code.

Exercise 3.1: Transactional TS.

Each **TS** must run in **a single transactional session of communication with the database**. It must be ensured that the Gateways invoked from a TS use, **ALL of them**, the same connection, to guarantee a single transaction. However, each operation invoked on a Gateway creates a connection. **La gestión de la conexión y la transacción debe hacerse en los objetos de negocio, de ahí su nombre transaction script**.

The general structure of a TS would be:

- Create the gateways you need .
    * AddMechanic necesita MechanicGateway
- Get connection to the database. This will involve modifying the gateways. A Jdbc class (additional code) is provided that takes care of creating a connection and saving it in the execution thread to retrieve it later at each gateway.
- Disable autocommit mode.
- Perform the relevant operations using, if necessary, the gateway(s) to retrieve the information or update it.
- Confirm the transaction.
- Close the connection.
- Return result (if any) to the user interaction layer. The data transfer objects between UI and BL remain the same (MechanicDto, InvoiceDto...).

In case of any exceptions during execution, rollback is necessary.

Exercise 4: Apply Command pattern in Service Layer

The organization of the business layer follows the Transaction Script pattern. This pattern organizes all the business logic as procedures that implement logic of a certain action requested by the user interface and access the database generating a single transactional session with database.
This pattern can be used in a more elegant way, if we use **Command pattern**.
Some of the roles of this pattern are already implemented:

- **Command interface**: The Command interface is provided. The TS must implement it.
- **Invoker**: A java class (JdbcCommandExecutor) is provided in the same folder. It is responsible for starting the execution of specific commands. It groups all actions common to all commands. However, it will be necessary to adjust our classes to suit others:
- **Client**: MechanicCrudServiceImpl class creates and configures command objects.
- **Concrete Command** and **Receiver**: In a single class in our implementation. Extract common code and move it to JdbcCommandExecutor.

Exercise 6: Repeat again with billing workorders use case

Later, as **homework**, you must repeat everything again with the code to **invoice workorders**. And once more with the **find not invoiced workorder by nif** use case.