Laboratory session 3: CWS1

Goal

- Split code in two layers: UI and service (the latter will include business and data access layer code)

Getting started

In Campus Virtual, there is a lot of code to start with:

- **Java Code Conventions** file.
- The **eclipse CWS0 project** and **cws_util**. The latter has some kinds of utility, which can be used freely but should not be modified**.**
- Folder with hsqldb database.
- Folder with additional code that must be used as needed, specifically:
  - Service interfaces (service → service interfaces) that include Data Transfer Objects (DTOs). These objects are used to pass information between different layers.
  - Assembler classes to transform some DTOs into others or ResultSet objects into DTOs.
  - Utility classes to display information in the screen (ui → cws_util), read configuration files (Conf).

HSQLDB server connection details:

- URL = "jdbc:hsqldb:hsql://localhost";
- USER = "sa";
- PASS = "";

The only agreed Java version is the one we work with in the laboratories.

Create a workspace and import projects CWS0 and cws_util. Then, modify the code in CWS0 to became CWS1 as you implement the following exercises. We will focus on the mechanics management use case (actor **manager/administrator**).

First, start your database and ensure that the corresponding driver is added to your project's build path. **Then, run the Main class in the package manager and try all the options in the mechanic management use case**. Use always the driver in the lib folder.

Unless otherwise stated, it is forbidden to rename classes, packages, methods, etc., either in this session or in the future. That includes capitalization. It is also disallowed to move classes from the package written in its "package" declaration or to change the hierarchy of packages that is derived from the "import" statements of the code.

1 Applying the layer pattern to separate user interface code from business and data access code

The code related to user interaction is split from the rest of the code, initially considering the use cases of mechanics management.

- Create a package structure with the code of the application, splitting into layers according to the **service layer pattern**; packages are added depending on the layers in which we are going to divide the application. The **uo.ri.cws.application** package will now contain two subpackages:
  1. **ui**. After some refactoring, these classes will contain just code referred to user interaction.
  2. **service**. It will contain classes that implement both business logic and persistence.

1.1 Organize business layer.

Service layer could be **organized** by **actors**, although in complex use cases, it could be done by **entities** in the domain model to increase **cohesion**.

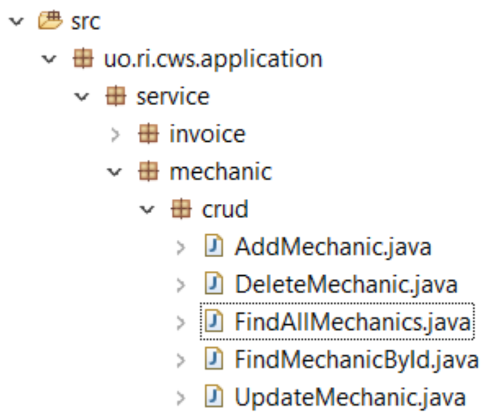
- Create a package for business objects (mechanic, invoice, ...). For example, **service.mechanic** will address mechanic management use case.
- Each of these packages shall, **as a rule**, conform to the following:
  - It will contain the service interface that includes the **data transfer objects (DTO) to encapsulate the parameters**, both input and output, that are exchanged **between this service and the clients**. These DTO objects will be named with a compound name of the entity name plus the suffix "Dto" (MechanicDto) and will be part of the corresponding service interface.
  - It will include a DtoAssembler class that will create the DTO by transforming data that comes from queries (ResultSet).
  - It will also enclose one or more packages in which the classes that will be responsible for carrying out the operations of the use case will be implemented.

Style standard: For those use cases that only have creation, update, recovery or deletion operations, such as mechanic management, a subpackage will be added named "**crud**".

- The classes in this **crud** package will generally meet the following requirements:
  - They will be called the same as the classes in the ui package (from which they will copy part of the code), removing the suffix "Action".
  - They will have a constructor without parameters or with a single parameter, which could be:
    - A DTO with the data to be added or updated.
    - A String with the identifier of the object to be deleted.
    - Depending on the recovery criteria, you may or may not have arguments.

IMPORTANT: The constructor should also check parameters and launch *IllegalArgumentException* if necessary. Unless otherwise denoted, empty or null parameters are considered invalid. There are classes in the cws_util project to perform these validations (eg. assertion.ArgumentChecks.java).

* They will also contain a single public method **execute()**, without parameters. It will implement the **use case operations**. The **returning type** will be:
  - Adding use cases: A DTO, containing all available information, including identifier, in adding.
  - Update or delete use cases: Void.
  - Listing use cases: Depending on the read criteria a single **DTO** with all available data or a **collection**.


```
src
  uo.ri.cws.application
    service
      invoice
      mechanic
        crud
          AddMechanic.java
          DeleteMechanic.java
          FindAllMechanics.java
          FindMechanicById.java
          UpdateMechanic.java
```

1.2 Implementing business layer.

– **Refactor** the classes ui.manager.action.* **taking all the code not handling user interaction or data validations** (NIF field length, email field structure, etc.) and **move it into the new classes** in service.mechanic.crud.
– The **UI classes will now invoke** the necessary classes in **the service layer** by passing as a parameter the data they need (in the form of a DTO, String).
– When implementing **execute**, be careful with possible business rule violations, such as deleting a non-existent mechanic. You can put it off until the end of this session. We will address them in the next session.

1.3 Implementing user interface layer.

– For printing mechanics, or other entities in the domain model, use Printer class provided (see extra code folder). Write new methods here as needed moving code from other classes if possible.

1.4 Test

Start database and **do the following** in the new application.

a. Create a new mechanic.
b. Update it.
c. Delete it.
d. Try to update a non-existent mechanic.
e. Try to delete a non-existent mechanic.
f. Display all mechanics.

2 Exercise 2: Using façade pattern to decouple UI layer from business layer.

The **façade pattern** will help to **remove dependencies** between user interface and business layers, providing each subsystem (mechanic management or other) with a **façade** (**interface**).

Clients (UI layer) will call methods in the facade instead of directly invoking methods in service classes (reduces coupling). Keep in mind these indications to introduce facades in your project:

– The additional code includes the java interfaces of the facades (see service → service interfaces).
– The façades provided cannot be modified (unless otherwise stated).
– For the mechanic management use case, the interface is **MechanicCrudService**.

2.1 Organize business layer.

– Create a new package **service.***.crud.commands**. Move here classes implementing operations (AddMechanic, DeleteMechanic, …).
– Check if all the Java interfaces are placed in its corresponding service packages. For instance, the interface MechanicService should be placed in the **service.mechanic** package**.**

2.2 Implement the facade

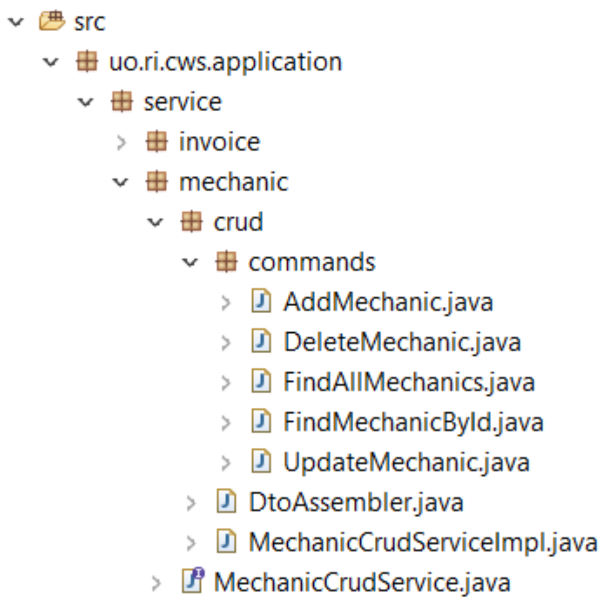
– It is necessary to implement the java interface, in the implementation package.

  Mechanic case: service.mechanic.crud

– The class will have the same name as the Java interface with the suffix “Impl”. Each of its methods will be responsible for invoking the corresponding deployment class

  Mechanic case: AddMechanic, DeleteMechanic, etc.

– Modify the ui (***Action) classes to invoke methods from the façade (uo.ri.cws.application.service.mechanic.crud.MechanicCrudServiceImpl), and not from the service implementation objects (AddMechanic, DeleteMechanic, ...).



```
src
  uo.ri.cws.application
    service
      invoice
      mechanic
        crud
          commands
            AddMechanic.java
            DeleteMechanic.java
            FindAllMechanics.java
            FindMechanicById.java
            UpdateMechanic.java
          DtoAssembler.java
          MechanicCrudServiceImpl.java
        MechanicCrudService.java
```

uo.ri.cws.application.service.mechanic.crud.commands

Style rules: The name of the java class that implements the façade will always be the same as that of the interface, adding the "Impl" suffix.

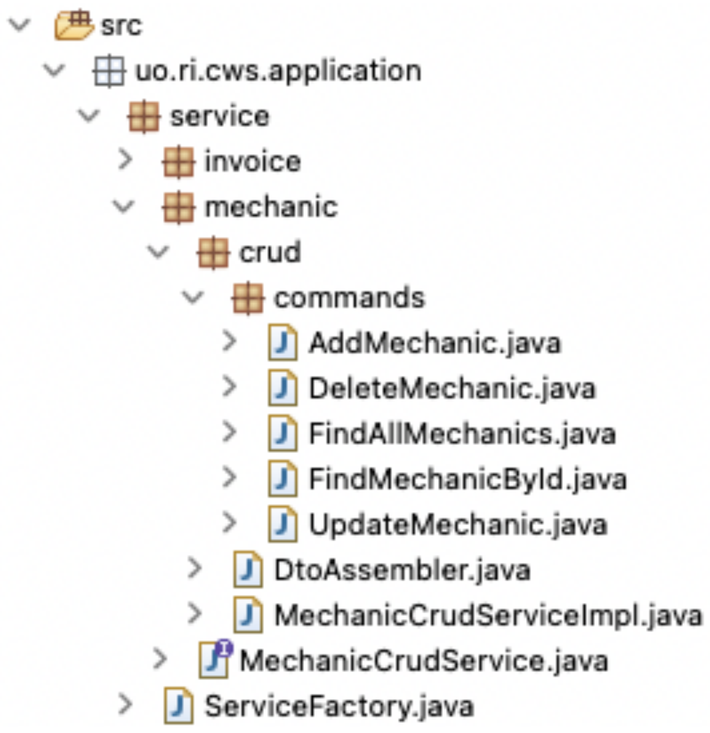
Style rule: If it is necessary to create new business objects, the name will be the same as the one of the interface method, except for the initial capitalization. For example, a findMechanicByAge method would result in a FindMechanicByAge class.

3 Exercise 3: Simple Factory (or Class Factory) pattern to decouple UI and business layers

With the Simple Factory pattern (https://javajee.com/factory-design-patterns-simple-factory-factory-method-static-factory-method-and-abstract-factory) the façade creation logic is hidden from the client. Instead of knowing the exact object class and instantiating it through a constructor, the responsibility of creating an object is moved away from the client into a factory object. This reduces coupling between clients and facades.

- The ServiceFactory **interface** provides methods for obtaining each of the façades (forMechanicCrudService(), forCreateInvoiceService(), etc.).
- Its implementation will be in the service package and will act as a façade factory and from it it will be possible to obtain instances of the facades of the services without knowing details of their implementation (decreases coupling).
- **The UI will use ServiceFactory to get the facades you need**.
  - Modify UI classes to remove dependencies on deployment classes at the business layer (facades, commands...).

* There will be no references to classes (MechanicServiceImpl), but to interfaces (MechanicService).



```
src
  uo.ri.cws.application
    service
      invoice
      mechanic
        crud
          commands
            AddMechanic.java
            DeleteMechanic.java
            FindAllMechanics.java
            FindMechanicById.java
            UpdateMechanic.java
          DtoAssembler.java
          MechanicCrudServiceImpl.java
        MechanicCrudService.java
      ServiceFactory.java
```

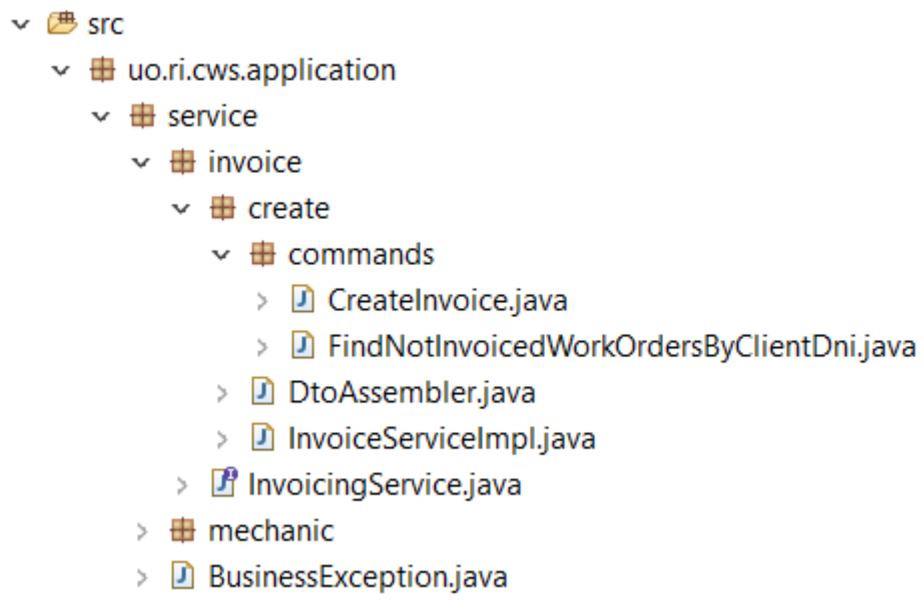
4 Exercise 4: New use cases. Create invoice for workorders.

From the solution to the previous exercises, the new use case Invoicing jobs must be implemented. This use case is already fully implemented in the WorkOrderBillingAction class in the uo.ri.cws.application.ui.cashier.action package.

Refactor the code and reapply the previous patterns to remove all non-UI code from the WorkOrderBillingAction class, bringing it to the required classes in the business layer.

It will be organized around the **Invoice entity**:

- The use case implementation package will be **uo.ri.cws.application.service.invoice**.
- The **InvoicingService** interface is provided with the **InvoiceDto** data transfer object that must be in the previous package.
- A new **package uo.ri.cws.application.service.invoice.create** is created to implement the create invoice use case.
- The **uo.ri.cws.application.service.invoice.create.commands** package will contain the implementation of the objects that perform the actions (**CreateInvoice**).



```
src
  uo.ri.cws.application
    service
      invoice
        create
          commands
            CreateInvoice.java
            FindNotInvoicedWorkOrdersByClientDni.java
          DtoAssembler.java
          InvoiceServiceImpl.java
        InvoicingService.java
      mechanic
      BusinessException.java
```

Repeat the exercise once again with the **Find Unbilled Jobs by Customer use case**. The implementation of this use case is currently located at **ui.cashier.action.*FindNotInvoicedWorkOrdersAction***.