Práctica 3: CWS1

Objetivos

- Separar el código en dos capas: interfaz de usuario y lógica de negocio (que en esta práctica incluirá las capas de lógica de negocio y persistencia).

Material necesario

En el Campus Virtual de la asignatura puede encontrar:

- El fichero **Java Code Conventions**.
- El proyecto eclipse **CWS0.**
  - una carpeta extra code con código adicional de uso obligatorio a medida que se necesite, concretamente:
  - Interfaces de servicios (business → services interfaces) que incluyen los Data Transfer Objects (DTOs). Estos objetos se utilizan como medio para pasar información entre distintas capas.
  - Clases assembler para transformar unos dtos en otros u objetos ResultSet en dtos.
  - Clases de utilidad para mostrar información por pantalla (ui → util), leer ficheros de configuración (conf).
- El proyecto Util con clases útiles (algunas usadas en CWS0) que no debe modificarse,
- una carpeta **data** con una base de datos hsqldb.

Los datos de conexión para el servidor HSQLdb:

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

La versión de Java utilizada será la instalada en los laboratorios de prácticas.

Cree un espacio de trabajo e importe en él los proyectos CWS0 y Util. A partir de este código inicial debe comenzar a implementar las prácticas. Nos centraremos en el caso de uso de gestión de mecánicos, cuyo actor es el administrador (gerente).

Antes de comenzar, inicie la base de datos y asegúrese de que el driver está en el path de su proyecto. Después, ejecute la aplicación de gestión de mecánicos, a partir de la clase MainMenu en el paquete manager. **Utiliza siempre el driver proporcionado en la carpeta lib.**

IMPORTANTE: a no ser que se indique explícitamente, está prohibido renombrar clases, paquetes, métodos, etc. Esto aplica tanto a esta sesión como a sesiones futuras relacionadas con el CarWorkShop. Tampoco está permitido mover clases a un paquete diferente del declarado en su sentencia "package" o modificar la jerarquía de clases implícita en las diferentes sentencias import del código proporcionado.

Ejercicio 1: Aplicación del patrón service layer

Se separa el código relativo a la interacción con el usuario del resto del código, considerando inicialmente el caso de uso de gestión de mecánicos.

- Crear una estructura de paquetes para contener el código de la aplicación, separando en capas según el patrón **service layer**; se añaden paquetes en función de las capas en que vamos a dividir de la aplicación. El paquete **uo.ri.cws.application** contendrá ahora dos subpaquetes:
    * **ui**. Se mantiene el paquete ui que contendrá ahora aquellas clases que implementen la interacción con el usuario.
    * **business**. Contendrá las clases que implementen tanto la lógica de negocio como el acceso a la base de datos. La clase de excepción BusinessException también debe moverse a este nuevo paquete.

1.1 Organización de la capa business

La capa **business** se organiza, como norma general, según **casos de uso. No obstante, en casos complejos como el nuestro, la organización se hará por entidades del modelo del dominio para incrementar la cohesión**.

- Se creará un paquete para cada entidad del dominio (mecánico, factura…). Por ejemplo, el paquete **business.mechanic** se ocupará del caso de uso gestión de mecánicos.
- Cada uno de estos paquetes se ajustará, **como norma general**, a lo siguiente:
    * Contendrá los **objetos de transferencia de datos**, **dto**, **para encapsular los parámetros**, tanto de entrada como de salida, que se intercambien **entre esta capa y la interfaz de usuario**.

      Estos objetos dto formarán parte de la interfaz del servicio correspondiente.
    * Incluirá un paquete **assembler** con una clase XXXAssembler (MechanicAssembler, InvoiceAssembler, etc) para procesar los dto.
    * Incluirá uno o varios paquetes en los que se implementarán las clases que se encargarán de llevar a cabo las operaciones del caso de uso.

Norma de estilo: Para aquellos casos de uso que sólo tengan operaciones de creación, actualización, recuperación o borrado, como gestión de mecánicos, el paquete se llamará **crud**.

- Las clases de este paquete **crud** se ajustarán, en general, a los siguientes requisitos:
    * Se llamarán igual que las clases del paquete ui (de las que copiarán parte del código), eliminando el sufijo Action.
    * Tendrán un constructor **sin parámetros o con un único parámetro**, que podría ser:

- Un DTO con los datos a añadir o actualizar.
- Un String con el identificador del objeto a eliminar.
- Dependiendo del criterio de recuperación, podrá o no tener argumentos.

IMPORTANTE: El constructor también debe hacer checkeo de parámetros y lanzar IllegalArgumentException en caso necesario. Si no se indica lo contrario, parámetros vacíos o null son considerados inválidos. Existen clases en el proyecto util para realizar estas validaciones.

* Un único método público ***execute***, sin argumentos, que **implementará las reglas de negocio,** y cuyo valor de retorno será:

- Casos de añadir objeto: Un DTO completo, que incluya el identificador del objeto añadido.
- Casos de borrado o actualización: void.
- Casos de lectura (búsqueda): un DTO o una colección de DTOs, según la naturaleza del método.

1.2 Implementación de la capa bussiness

- **Extraer de las clases ui.administrator.action.XXX** todo aquel código que no se refiera a interacción con el usuario o pequeñas comprobaciones de validez de datos (longitud del campo nif, estructura del campo nif o email, etc) y mover ese código relativo a lógica de negocio o acceso a datos a las **nuevas clases del paquete business.mechanic.crud**.
- Las clases de la **interfaz invocarán** ahora a las clases necesarias en **la capa de negocio** pasando como parámetro los datos que necesiten (en forma de dto).
- Además de implementar la lógica de negocio necesaria para llevar a cabo una acción, el método **execute()** de las clases en paquetes **crud** también deberían realizar validación de reglas de negocio. *Por el momento*, se ignorarán posibles violaciones de las reglas de negocio, como añadir un mecánico repetido o borrar uno que no exista. **Recuerda implementar estas comprobaciones al final de la sesión.**
- Para imprimir los mecánicos (u otras entidades del modelo de dominio), se debe usar la clase **Printer** proporcionada (ver código adicional). Escribe nuevos métodos según se necesiten, moviendo código de otras clases si es posible.

1.3 Test de funcionamiento

Arranque la base de datos y realice las siguientes **pruebas**

a. Crear un nuevo mecánico
b. Modificar el mecánico anterior
c. Borrar el mecánico anterior
d. Modificar un mecánico cuyo id no existe
e. Intentar borrar un mecánico que no existe
f. Listar los mecánicos.

2 Ejercicio 2: Aplicación del patrón fachada para desacoplar el código de las capas ui y business

Para **eliminar dependencias** entre la capa de interfaz de usuario y la de negocio, se aplica **inicialmente** el patrón **fachada**, dotando a cada subsistema (gestión de mecánicos u otro) de una **interfaz** (**fachada**) que actúa como **punto único de entrada**. Los clientes (capa ui) invocan métodos de la fachada; no acceden directamente a los objetos del subsistema (disminuye el acoplamiento). Tenga en cuenta estas indicaciones para introducir fachadas en su proyecto:

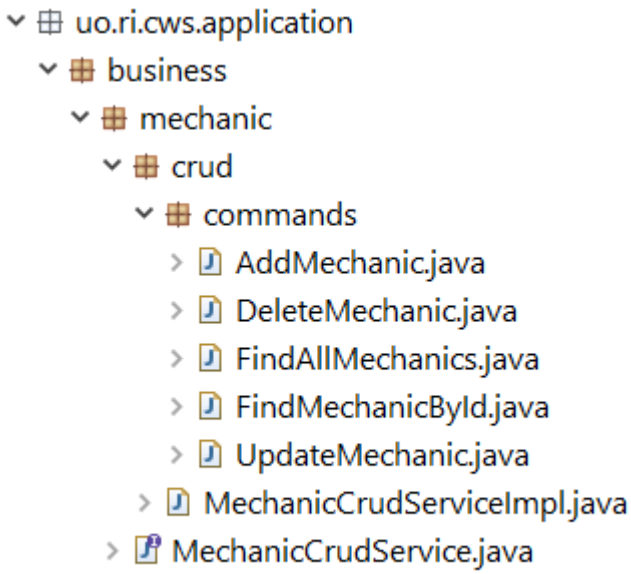
- EL código adicional incluye las interfaces java de las fachadas (ver business → services interfaces).
- Las fachadas proporcionadas no se pueden modificar (salvo que se indique lo contario).
- Para el caso de uso de gestión de mecánicos, el interfaz es **MechanicService**.

2.1 Organización de la capa business

- Crear un nuevo paquete **business.XXX.crud.commands** y mover a él las clases que implementan las distintas operaciones (AddMechanic, DeleteMechanic, ...).
- Colocar las interfaces java de las fachadas en el subpaquete correspondiente del paquete business.
- La interface **MechanicService** se sitúa en el paquete **business.mechanic**

2.2 Implementación de la fachada

- Es necesario implementar la interface java, en el paquete de implementación.
  * Caso de mecánico: business.mechanic.crud
- La clase tendrá el mismo nombre que la interface Java con el sufijo **Impl**. Cada uno de sus métodos se encargará de invocar a la clase de implementación correspondiente
  * Caso de mecánico: AddMechanic, DeleteMechanic, etc.
- Modifique el cliente (***Action) para que invoque métodos de la fachada, y no de los objetos de implementación del servicio.



Normas de estilo: El nombre de la clase java que implementa la fachada será siempre el mismo que el de la interface, añadiendo el sufijo **Impl**.

Norma de estilo: Si fuese necesario crear nuevos objetos de negocio, el nombre será exactamente igual que el nombre del método de la interfaz, excepto la mayúscula inicial.

3 Ejercicio 3: Aplicación del patrón Simple Factory (o Class Factory) para desacoplar el código de las capas ui y business

El patrón Simple Factory (https://javajee.com/factory-design-patterns-simple-factory-factory-method-static-factory-method-and-abstract-factory) permite que se puedan obtener instancias de las fachadas sin conocer detalles de su implementación (disminuye el acoplamiento). Una clase separada encapsula la creación de objetos.

- Se proporciona la clase java **BusinessFactory** que actúa como fábrica (factoría) de fachadas y de la que se puedan obtener **instancias** de las fachadas de los servicios sin conocer detalles de su implementación (disminuye el acoplamiento). Esta clase contiene mucho código comentado. Puede descomentarlo a medida que sea necesario.
- BusinessFactory ofrece un método para obtener cada una (forMechanicService(), forInvoicingService(), etc.).
- **La interfaz de usuario utilizará BusinessFactory para obtener las fachadas que necesite**.
    * Modifique las clases de interfaz de usuario para eliminar las dependencias de las clases de implementación en la capa de negocio (fachadas, commands…).
    * No habrá referencias a clases (MechanicServiceImpl), sino a interfaces (MechanicService).

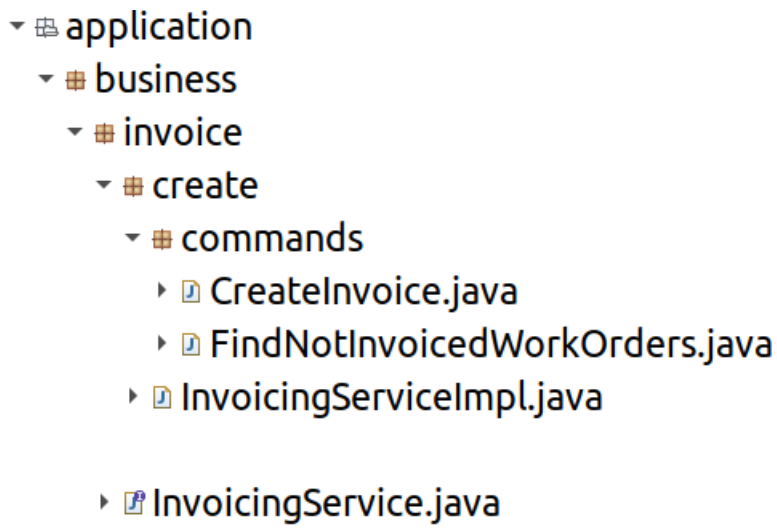
4 Ejercicio 4: Nuevos casos de uso. Buscar trabajos no facturados por cliente y Facturar trabajos realizados en el taller generando la factura

A partir de la solución a los ejercicios anteriores, se debe implementar el nuevo caso de uso Facturar trabajos. Este caso de uso está ya completamente implementado en la clase WorkOrderBillingAction en el paquete uo.ri.cws.application.ui.cashier.action.

Refactorice el código y vuelva a aplicar los patrones anteriores para eliminar de la clase WorkOrderBillingAction todo el código que no se refiera a interfaz con el usuario, llevándolo a las clases necesarias en la capa de negocio.

Se organizará en torno a la entidad **Invoice**:

- El paquete de implementación del caso de uso será **uo.ri.cws.application.business.invoice**.
- Se provee la interfaz **InvoicingService** con el objeto de transferencia de datos **InvoiceDto** que deberán estar en el paquete anterior.
- Se crea un nuevo paquete **uo.ri.cws.application.business.invoice.create** para implementar el caso de uso de crear factura.
- El paquete **uo.ri.cws.application.business.invoice.create.commands** contendrá la implementación de los objetos que lleven a cabo las acciones (**CreateInvoice**).



Se recomienda repetir el ejercicio una vez más con el caso de uso Buscar trabajos no facturados por cliente. La implementación de este caso de uso se encuentra actualmente en **ui.cashier.action.*FindNotInvoicedWorkOrdersAction***.