Checklist para la entrega de la práctica de JDBC

Esta checklist no es una rúbrica ni un contrato de evaluación con la que podrás determinar de forma precisa la nota de tu entrega. Es simplemente una lista de los errores más frecuentes en estas entregas. La intención es que la uses antes de subir al campus tu proyecto.

Este documento acompaña a la hoja Excel que debes rellenar y adjuntar a la entrega de tu trabajo. No olvides entregarla cumplimentada.

Los chequeos de errores que se comentan a continuación son algunos de los más frecuentes detectados a lo largo de varios cursos, pero no son todos. No tener ninguno de estos no te garantiza que tu entrega sea perfecta, pero si será bastante mejor que si no la revisas.

Precondiciones
Las interfaces de la capa de servicio no han sido modificadas Los test de aceptación originales se compilan y ejecutan correctamente Estas tres precondiciones deben cumplirse para que la entrega no sea descartada. Las interfaces deben ser respetadas tal cual se han entregado. Constituyen un contrato entre el cliente y el proveedor del servicio. Ten en cuenta que un script copiará las interfaces y los tests originales en tu entrega y después ejecutará los tests. Por lo que las modificaciones que hayas hecho a tus tests e interfaces se perderán y se generarán errores de compilación y/o ejecución, lo que invalidará la entrega. Antes de hacer tu entrega verifica esto, no vaya que a ser que, inadvertidamente, hayas introducido algún cambio en el código de los tests o de las interfaces.
Documentación
Sin fichero readme.txt En este fichero debes especificar tu nombre, los casos de uso que te han tocado y cualquier otro comentario que consideres. No entregarlo está penalizado. No se ha entregado el modelo de dominio ampliado en la tarea correspondiente y en la fecha indicada.
Código
Con warnings En código para entregar no puede haber warnings. Un warning es un aviso del compilador de posibles errores. Entregar código con ellos genera inseguridad en el que lo recibe y transmite sensación de código chapucero y descuidado. Todos los warnings pueden y deben ser evitados, bien cambiando el código de forma que se evite la potencial fuente de error o, en casos muy retorcidos, silenciándolo con @SupressWarnings (usa este último con responsabilidad). No se siguen recomendaciones Java Code Conventions Los convenios de nombrado, sangrado, etc., son importantes para que en una comunidad de programadores se pueda compartir código. También dan indicaciones de cómo escribir el código de forma que se entienda su estructura de forma rápida. Todos los lenguajes tienen el suyo. En Java se sigue como referencia básica las “Java Code Conventions”. En el enunciado se menciona expresamente que el código debe ajustarse a este convenio. Para la evaluación de este punto se prestará atención a los nombres de atributos, métodos y variables, al empleo del camelCase y al correcto sangrado del código. Una fuente habitual de problemas de nombres suele ser forzar el nombre de atributos Java (camelCase para nombres compuestos) para que coincida con el nombre de una columna en la tabla (donde se suele usar el _ para nombres compuestos y mayúsculas).

Líneas de código demasiado largas

Las JCC recomiendan que no se pase de la columna 80. Aunque el límite te parezca escaso hay razones para ello.
Si configuras el IDE te lo puede resolver medianamente bien cuando le dices que formatee el código, pero tienes que decírselo tú. Y no es perfecto. Por ejemplo, si has puesto algún formateo particular en los comentarios te lo quitará y el estilo "fluent interfaces" no lo suelen respetar. La forma de evitarse problemas es respetar los márgenes de la escribes el código.
No se aplicará esta penalización por unas pocas líneas que pasen 3 o 4 caracteres del límite, pero sí cuando sea evidente que se está ignorando esta recomendación.

Nombres de clase incorrectos (plurales, tablas, etc.)

Los nombres de clases van en singular, ya que una clase representa un concepto. Por el contrario, los nombres de las tablas suelen ir en plural, ya que se interpretan como "todas esas <cosas> que se guardan ahí". También se considera un error nombrar las clases como las tablas para así evitar tener que adaptar el mapeo con más anotaciones.

Código mal sangrado

El sangrado de código es importante para captar la estructura del código de un simple golpe de vista.
Ejemplos como este no permiten esa percepción:

```
    private Order order;

    public ReceiveOrder(Pedido order) {
       this.order = order;
    }

    @Override
    public Object execute() throws BusinessException {
    order.receive();

    order.setReceptionDate(new Date());
    order.markAsReceived();

    for (OrderLine rp : order.getOrderLines())
        Jpa.getManager().merge(rp.getRepuesto());

    order = Jpa.getManager().merge(order);

    return order;
    }

    }
```

Caso especial son las partes de código en las que se emplee el estilo interfaces fluidas[1].
Mantén la alineación vertical.
Esto está bien:

```
@Override
public Optional<Xxxxx> findByCode(String code) {
   return Jpa.getManager()
      .createNamedQuery("Xxxxx.findByCode", Xxxx.class)
      .setParameter(1,  code)
      .getResultList().stream()
      .findFirst();
}
```

Esto está mal:

```
@Override
public Optional<Xxxxx> findByCode(String code) {
   return Jpa.getManager()
      .createNamedQuery("Xxxxx.findByCode", Xxxx.class)
      .setParameter(1,  code).getResultList().stream().findFirst();
}
```

Métodos no vacíos con comentarios //TODO

Dan sensación de código descuidado. En una entrega se deja código limpio.

[1] https://martinfowler.com/bliki/FluentInterface.html

Métodos con código comentado

Dan sensación de código descuidado. En una entrega se deja código limpio.

Algunos bugs importantes

Como la presencia de campos no privados en las clases.

Algunos bugs menores

Pequeños errores de programación por mal uso del lenguaje o de los idioms habituales.

catch NullPointerException

Se considera mala práctica. Una excepción de este tipo indica un error de programación, es decir, si se produce es porque nuestro código tiene bugs. Lo que se debe hacer es arreglar el código para que no lo produzca, pero no tratar ese tipo de problema.

catch Exception

Se considera otra mala práctica. Al atrapar *Exception* se tratan por igual todos los posibles tipos de error que puedan aparecer. Al menos se debe distinguir entre errores de lógica de negocio (quizá culpa del usuario) y errores de sistema/programación. El tratamiento genérico debe ser diferente. A menudo unos paran el programa, otros no.

Negocio

Fachada de servicios con lógica, fuera de lugar

La misión de la fachada de la capa de servicio es ocultar las clases que hay detrás, pero no ejecutar lógica de negocio. Simplemente reparte, o dirige a la clase adecuada, la responsabilidad de resolver el servicio que corresponda. Si se le añade lógica pierde cohesividad (principio básico de diseño) y pasa a ser totum revolutum con, posiblemente, cientos de líneas de código y se hace difícil de mantener.

Lógica mal implementada

Si te pasan los tests es difícil que pueda darse este caso, pero es posible.

Transaction scripts no implementan la interfaz Command

Todos nuestros objetos de servicio (o negocio) se implementan siguiendo los patrones Transaction Script (TS) y Command. Para el segundo es necesario que implementen la interfaz Command proporcionada.

Los comandos no se ejecutan utilizando el JdbcCommandExecutor

Los comandos manejan conexiones y/o transacciones en lugar del executor

Si esto sucede, revisa el punto anterior. Están muy relacionados.

Los comandos lanzan excepciones SQL, Persistence o Runtime

Los comandos sólo deberían lanzar BusinessException o IllegalArgumentExceptions.

Los objetos de negocio (TS) tienen dependencias de la capa de presentación o de otros objetos de negocio

En la arquitectura en capas sólo se permiten dependencias de la capa inmediatamente inferior. Por tanto, revisa que en un TS no haya dependencias de paquetes de ui o de otros objetos de la capa service. Recuerda que nuestros transaction scripts están implementados en paquetes command por lo que no deberías tener sentencias import de otros command, fachadas o factoría de servicio.

Comando imprime en consola

Un comando se ejecuta en la capa de servicio. Sólo puede mostrar información al usuario la capa de presentación. Hacer eso en un comando supone no haber entendido el diseño general de la aplicación.

Si has puesto *System.out.println()* para hacer trazas o comprobaciones mientras desarrollabas acuérdate de quitarlos, cuando entregues no vamos a poder saber con qué

intención lo pusiste. Será más difícil que se te olviden si en vez de en *System.out* imprimes en *System.err*.

Algún comando no se ajusta al Javadoc de la interfaz

El Javadoc de las interfaces especifica los resultados esperados y las excepciones que se deben lanzar. Se considerará este error si no se están verificando todos los chequeos que se especifican o la lógica no se resuelve como se indica, aunque la operación funcione en algunos escenarios.
Te aconsejo que, según vayas implementando el código de negocio tengas siempre delante la interfaz de servicio. Ahí está descrita a grandes rasgos la lógica que debe implementarse en la capa service y cómo.

catch BusinessException en comando incorrecto

Se considera este error cuando en el código de un comando se atrapa una BusinessException y el tratamiento que se le da es incorrecto. Por ejemplo: se silencia, se imprime en pantalla e.printStackTrace(), o se imprime en consola un error.

Los TS realizan cambios de objetos Record a objetos Dto innecesariamente

La información que se obtiene de la capa de persistencia llega a los TS en forma de objetos Record (Dto de la capa de persistencia). La capa service puede trabajar con esa información. Crear un objeto Dto sólo es necesario con el fin de retornar información a la capa de presentación.

Los objetos command no utilizan la factoría de persistencia para utilizar los gateways

Para desacoplar las capas de servicio y persistencia se aplica el patrón factory que consiste en ocultar en una factoría la creación de instancias de los gateways. En lugar de eso, crear esas instancias en los objetos de negocio es un error.

Persistencia

Métodos de gateway con lógica de negocio

La implementación de persistencia tiene como única misión ocuparse de traer y llevar objetos de la base de datos. No le corresponde analizar esos datos, validarlos, ni cuestionar los resultados de las consultas, ni generar códigos únicos. Son simplemente operaciones CRUD puras y duras.

Métodos de clases de persistencia lanzan BusinessException

Generalmente con métodos del estilo:

```
List<…> res = statement.executeQuery(…);


if (res.size() == 0) {
       throw new BusinessException("Does not exist...");
}
```

La capa de persistencia no tiene el conocimiento suficiente como para decidir que el resultado de una consulta, o su ausencia, sea un error de lógica de negocio. Si es un error de lógica de negocio se decidirá en la capa de más arriba. Y si tampoco allí se puede decidir, será en la capa de presentación (quizá sea culpa del usuario, si ha tecleado un código que no existe…).
Este punto está relacionado con el de “Métodos Gateway con lógica de negocio”, pero en esta ocasión se pone énfasis en el tipo de excepciones que se están lanzando.
Las BusinessException solo las lanza el paquete application.service.

Métodos de clases de persistencia lanzan SQLException en lugar de PersistenceException

Con el fin de eliminar la dependencia de la base de datos, el Gateway lanzará una excepción genérica PersistenceException y no SQLException. Un método de la capa persistence, sólo lanzaría PersistenceException (no chequeada), lo que indicaría algún tipo de error de sistema o de programación: errores del tipo integridad referencial, problemas de conexión a la base de datos, errores de sintaxis o semánticos en las consultas, etc. En todo caso errores de sistema o bugs.
PersistenceException en no chequeada ya que hereda de RuntimeException.

Los objetos Gateway crean y liberan conexiones y/o transacciones

La gestión de las transacciones (y por tanto de las conexiones) se realiza en los transaction script de la capa service. De esta forma, varios Gateway pueden compartir la misma conexión y ejecutarse en una misma transacción.

Los objetos Gateway no liberan los recursos que crean

Concretamente, las Statement o PreparedStatement y los ResultSet que hayan creado durante su ejecución.

Consultas SQL complicadas

Existen consultan que implementan lógica de negocio, como generar o recibir un pedido, calcular el importe de una factura en un select, etc

Insuficientes consultas

Indica que se hacen en Java el filtrado de datos que podría hacerse más fácilmente con una consulta SQL. En vez de ejecutar una consulta específica, se hace en el comando una llamada al método "findAll()" y luego se filtra programáticamente. Eso supone una pérdida de rendimiento y un uso excesivo de memoria. Sirve el ejemplo siguiente como muestra:

```
@Override
public List<ProviderDto> execute() throws BusinessException {
    List<ProviderDto> providers = new LinkedList<>();
    Optional<SparePartRecord> ospr = spg.findByCode(code);
    String id = ospr.get().id;
    List<SupplyRecord> srs = sg.findAll();
    for (SupplyRecord sr : srs) {
        if (sr.sparePartId.equals(id)) {
            providers.add( DtoAssembler.toDto(
                    pg.findById(sr.providerId).get()));
        }
    }
    return providers;
}
```

Otra forma de caer en este error es hacer métodos findBy… en los gateway muy complejos (con lógica en Java). En vez de hacer una query refinada se hace una query "gruesa" y en Java, en el método del gateway, se hace el filtrado.

Consultas incoherentes

Consultas que no devuelven los objetos que se suponen deben devolver. Consultas en gateways que devuelven objetos de otras tablas y no de la tabla asociada con el Gateway.

Hay consultas o parámetros de conexión no externalizados a los ficheros de propiedades

Deben estar todas, revisa que no se tu cuele alguna en un gateway.

Presentación

Hay lógica de negocio en las classes Action

Otra manifestación de que no se ha entendido la arquitectura. La presentación SOLO se ocupa de la interacción con el usuario. Punto. Si para actualizar un suministro se lleva a la capa de presentación el proveedor, el repuesto, el suministro, se cambian los precios etc, se está trasladando la responsabilidad a la capa de presentación. En este caso, en presentación se pediría el id del proveedor, el del repuesto y el nuevo precio. La capa de lógica se ocupará del resto.

Mover la lógica a la capa de presentación presenta varios problemas:

- Se rompe la cohesividad de los paquetes (hay una fuga de funcionalidad desde negocio a presentación) y se aumenta el acoplamiento.
- Se genera más trasiego de datos entre las capas del necesario. Puede llegar a ser crítico cuando la capa de presentación y la de lógica se ejecuten en máquinas separadas. No es el caso, de momento …
- Cada llamada a la capa de servicio se ejecuta en una transacción separada. Si se necesita hacer varias modificaciones cada una de ellas se harán de forma separada y no existirá la posibilidad de rollback() en caso de errores. Habrá que

añadir código extra para deshacer expresamente todas las modificaciones intermedias.
- Si hubiera que añadir otra capa de presentación (por ejemplo, web además de escritorio), habría que reescribir la misma lógica de negocio. Copiar y pegar o replicar. Lo que conllevaría problemas de mantenimiento, etc…

La capa de presentación no usa la factoría de servicio para utilizar las fachadas

Para desacoplar las capas de servicio y presentación se aplica el patrón factory que consiste en ocultar en una factoría la creación de instancias de las fachadas de servicio. En lugar de eso, crear esas instancias en los objetos de presentación (action) es un error.

catch BusinessException en Action

No se considera error grave, pero es una redundancia. Con la implementación de BaseMenu ya se atrapan estas excepciones con el propósito de mostrar en pantalla un mensaje. Eso ya lo hace la clase BaseMenu en su bucle principal.

catch PersistenceException o Exception en Action

Fuera de lugar. Atrapar esa excepción explícitamente implica acoplar la capa de presentación con la de persistencia, que debería estar oculta para ella tras la capa de servicio. Una PersistenceException indica un error de sistema o programación en la capa de persistencia que no tiene causa en el usuario ni en la lógica del negocio. Probablemente el usuario no podrá hacer nada. El programa termina y emite un mensaje de error.
Atrapar esa excepción con el propósito de mostrar un mensaje en pantalla y permitir que el programa continúe, indica no haber entendido como se gestionan las excepciones en el programa.

Ejecución

La aplicación no se inicia

Tiene errores de compilación, se ha entregado con una versión incorrecta de Java (debe usarse la que hay en los ordenadores de los laboratorios, Java 17), etc.
Indica que ni siquiera se ha probado la aplicación para entregar, o que se entrega a sabiendas (“a ver si cuela”), haciendo a los demás perder el tiempo que no se ha dedicado a la asignatura.

Incompleto, algunos casos de uso no se implementaron o no hacen nada

Se ha entregado una implementación que en la que faltan los comandos que implementan los casos de uso o se han dejado muy incompletos. Se entrega “a ver si cuela”…

La ejecución se interrumpe con excepciones

Revienta y muestra en pantalla una traza de excepción en situaciones como por ejemplo:
- Al teclear códigos que no existen.
- Al generar pedidos para piezas que ya están pendientes en otros pedidos
- Al borrar elementos que no pueden ser borrados.

Todos son errores de programación y probablemente por insuficiente validación en los TS..

Funciona con errores

Aunque se ha implementado toda la funcionalidad pedida y no lanza excepciones hay errores lógicos. Algunos ejemplos son: generar varios pedidos para una misma pieza bajo stock, no calcular correctamente los precios (errores de bulto, no pequeños decimales), etc.