Checklist para la entrega de la práctica de JPA

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 tu 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.

Aprovechamos también el documento para añadir posibles ampliaciones que harán subir la nota (puede pasar de 10) y compensar posibles fallos.

Precondiciones
Las interfaces de la capa de servicio no han sido modificadas Los tests originales compilan Los test originales se 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. Un script copiará las interfaces y los tests originales en la entrega y después ejecutará los tests. 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.
Ampliaciones
Hacer otros casos de uso además de los que te corresponden Se consideran válidos si la implementación es correcta. Añadir más tests Pueden ser de persistencia, de la capa de servicios o de métodos extra que añadas al modelo de dominio. Para ser considerada esta ampliación debe haberse añadido una cantidad significativa de test que verifiquen más allá de los casos sencillos. Se esperan por lo menos 10 métodos de test. Pasar todas las anotaciones @ Jpa al *orm.xml* Añadir anotaciones al código de las clases del modelo es muy intrusivo. Hace que el código quede muy contaminado con cosas que le son ajenas y además introduce dependencias de JPA (infraestructura) rompiendo en cierto modo el principio arquitectónico de que el modelo no tiene dependencias. Se propone como ampliación extraer todos los metadatos de mapeo al *orm.xml* y limpiar las clases del modelo del dominio. La estructura semántica de las etiquetas XML es la misma que la de las anotaciones y usando un editor avanzado de XML (como el de Eclipse) la operación no resulta complicada. Usar otro mapeador Cambiar el mapeador requeriría muy pocos ajustes. Otros posibles mapeadores podrían ser Hibernate, OpenJpa, etc. Se consideraría también válido implementar la persistencia con una base de datos orientada a objetos como ObjectDB, que soporta la especificación JPA.
Documentación
Sin fichero readme.txt En este fichero debes especificar tu nombre, los casos de uso que te han tocado, las ampliaciones que hayas hecho y cualquier otro comentario que consideres. Si has hecho ampliaciones y no las mencionas aquí no serán tenidas en cuenta.

No entregarlo está penalizado.

Sin UML de modelo de dominio

Sin diagrama de tablas

Son documentos pedidos expresamente en el enunciado. Su no entrega invalida la entrega.

Diagrama de tablas no hecho con ingeniería inversa

Se pedía diagrama hecho de forma automática con alguna herramienta. NO sirve diagrama "dibujado", aunque sea con alguna herramienta. Se busca un diagrama "fiel" del modelo de tablas para el que se hace el mapeo.

Diagrama UML hecho con ingeniería inversa

No sirve uno hecho con ingeniería inversa ya que queda a nivel de implementación y cosas como las asociaciones, o clases asociativas pierden su significado.

Modelo UML incompleto, incorrecto

Por algún motivo el diagrama es incorrecto. Por ejemplo, no coincide con las clases de implementación, no refleja todas las asociaciones, alguna asociación no tiene nombre y/o multiplicidad, etc.

Modelo relacional incorrecto

Por algún motivo el diagrama es incorrecto. Quizá no coincide con las tablas reales, faltan relaciones, etc.

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 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 (se suele usar el _ para nombres compuestos y mayúsculas) (emplea @Column(name="…") para ello).

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.

Mal nombres de clases (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 Pedido order;

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

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

    order.setFechaRecepcion(new Date());
    order.markReceived();

    for (RepuestoPedido rp : order.getRepuestosPedido())
        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] (API de queries Jpa y streams…). 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.

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. Unos paran el programa, otros no.

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

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.

Comando(s) con mucha lógica

La esencia del patrón modelo de dominio (idea central de toda esta parte de la asignatura) es que toda la lógica que pueda estar en las clases del modelo ahí debe estar. Por lo tanto, poner la lógica en comandos contraviene esa idea e invalida muchas de las virtudes de este patrón, dejando de nuevo los comandos como Transaction Scripts.

Un síntoma de este fallo es que los comandos son muy largos. Como referencia, la mayoría de los métodos *execute()* será de menos de 10 líneas. Puede que alguno te salga más largo, pero en ese caso comprueba que no se replica lógica que ya está en el modelo o que estaría mejor allí.

En general, un comando se limita a coordinar la acción de repositorios, crear o modificar objetos del grafo, invocar a sus métodos de lógica y realizar chequeos.

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.

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.

catch NoResultException en comando

Esta excepción salta por usar el método *getSingleResult()* para ejecutar una consulta que por su naturaleza siempre devolvería un resultado (por ejemplo, “select count(a) from …”) pero que en esa ejecución no devuelve ninguno (o más de uno).

En cualquier caso, indica un error de programación. Es una inconsistencia que no haya resultado, algo que es imposible que pase y sin embargo está pasando. Es un error del mismo tipo que romper una integridad referencial en la base de datos. Es un aserto que se incumple. Por lo tanto, sólo queda arreglar el código ya que es un bug en el programa. Añadir un manejador para esa excepción indica que no se entiende la esencia de ese método.

Además, acopla el comando con la tecnología de persistencia ya que esa excepción es propia de JPA y el comando debe permanecer ajeno a la tecnología de implementación de la persistencia.

Persistencia

Métodos de repositorios con lógica de negocio

La implementación de persistencia tiene como única misión ocuparse de traer y llevar objetos del grafo al soporte persistente. 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.

Mal uso de *getSingleResult()*

Se considera erróneo ejecutar este método sobre consultas en las que exista la posibilidad de que no devuelvan nada.
Por ejemplo, todas las del estilo

```
"select c from Client c where c.name = ?1"
"select m from Mechanic m where m.dni =  ?1"
```

…

Si no hay ningún cliente registrado con el nombre que se pasa, cosa perfectamente posible, la ejecución reventará con una NoResultException (no chequeada).

Métodos de clases repositorio lanzan BusinessException

Generalmente con métodos del estilo:

```
List<…> res = Jpa.getManager()
        .createNamedQuery( ... )
        .setParameter( ... )
        .getResultList();

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 de clases repositorio 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. Un método de repositorio, 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.

Consultas JPQL complicadas (a la SQL)

Se han usado consultas Theta-style, al modo más típico de las consultas SQL que no usan la cláusula JOIN (join en el where).
La misma consulta se podría haber redactado de forma mucho más compacta usando los join implícitos o el join de JPQL. Además la ejecución podría ser más eficiente (dependiendo de la BDD).
Indica desconocimiento de las posibilidades de JPQL.

Insuficientes consultas

Indica que se hacen en Java el filtrado de datos que podría hacerse más fácilmente con una consulta JPQL. 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 Object execute() throws BusinessException {
   List<Repuesto> res = new LinkedList<>();
   List<Repuesto> repuestos = repo.findAll();
   for(Repuesto r: repuestos) {
      if ( r.getSuministro().getProveedor().getName().contains( name )) {
          res.add( r );
      }
   }
   return r;
```

Otra forma de caer en este error es hacer métodos findBy… en los repositorios 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 repositorio, se hace el filtrado.

Demasiadas consultas

Al contrario que en el caso anterior, tener excesivas consultas indica que las clases del modelo del dominio no tienen lógica y/o que no se usa la navegación del grafo, lo que implica no haber entendido el patrón central de esta parte de la asignatura: patrón modelo de dominio. Un ejemplo de esto sería tener un mecánico en el contexto de persistencia y para obtener sus intervenciones ejecutar una query en el repo, cuando lo más sencillo y acceder a la colección de intervenciones directamente con *m.getInterventions().*
También puede indicar que se ha tratado de hacer un *porting* de la entrega anterior a esta.

Consultas incoherentes

Consultas redactadas en el *orm.xml* que no devuelven los objetos que se suponen deben devolver.

Hay consultas no externalizadas al *orm.xml*

Deben estar todas, revisa que no se tu cuela alguna en un repositorio.

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 y se establecen asociaciones 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…

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.

Modelo

Lógica mal implementada

Si te pasan los tests es difícil que pueda darse este caso, pero es posible. Si has añadido otros métodos públicos y no tienen tests, es posible que se te cuelen bugs. Hazles tests (que además te puede ayudar a subir puntos) o revísalos de nuevo.

Asociaciones mal establecidas en constructores de clases asociativas

En el caso de clases asociativas, los dos objetos a los que se asocia, y que forman la identidad, se pasan al constructor y es misión del constructor establecer esa asociación. En nuestro caso se debería llamar al método *link()* de la clase de asociación adecuada. Aunque luego no genere errores de ejecución, no es correcto asignar las referencias de la asociación en el constructor sin llamar a esa clase ya que no se cierran los correspondientes bucles de referencias.

Clases de modelo imprimen en consola

Es muy similar al caso de "Comando imprime en consola". Elimina todos los *System.out.println* antes de entregar.

Se lanza excepción impropia (o mal gestionada)

Las clases del modelo del dominio validan precondiciones y cuando alguna no se cumple se lanza IllegalArgumentException o IllegalStateException. Sin embargo, no lanzan BusinessException ya que esa excepción es de Business y se invertiría la dependencia entre esos paquetes. Los tests te valiarán que se lanzan las que se esperan, pero si has añadido más métodos públicos comprueba que lanzan.

Constructores no se encadenan

Cuando una clase tiene varios constructores, uno de ellos se establece como el fundamental, y los demás, los derivados, deben invocar al constructor básico en vez de reproducir de nuevo su funcionalidad, aunque sea mínima. No hacerlo es equivalente a esparcir código (y defectos) copiando y pegando.

Setters para atributos de identidad

Los atributos que definen la identidad de las entidades del modelo del dominio se establecen en el constructor y son de sólo lectura. Por lo tanto no pueden llevar setters.

Hay getters de atributos naturales o colecciones que devuelven mutables

Los atributos naturales mutables (ej. Date) y las colecciones no deben ser devueltos directamente en getters de entidades. Se debe devolver una copia para no comprometer la integridad del objeto del que forman parte.

Sin métodos *hasCode()*, *equals()*

Supone hacer una implementación parcial del modelo del dominio.

hashCode(), *equals()* mal definido

Debe ir definido sólo sobre los atributos que definen la identidad: Errores habituales:

- No definido sobre los atributos que forman la identidad de la entidad.
- Definido sobre más campos que los que forman la identidad.
- Definido sobre el campo @Id cuando es generado por el mapeador o la base de datos

Código de *link(), unlink(), _get*(), _set*()* incorrecto

Métodos *_get*()*, *_set*()* dejados como públicos, implementación de *link()* y *unlink()* que no siguen el orden adecuado, etc.

Repositorios invocados desde el modelo de dominio

Las clases del modelo del dominio no pueden tener dependencias de ninguna otra clase fuera del modelo del dominio. Eso permite, entre otras cosas, testear la lógica de negocio de forma aislada. El resto de capas sí tienen dependencias de esta capa, pero nunca a la inversa.

Llamar a repositorios (persistencia), para generar un nuevo código (per ejemplo, el siguiente número de factura) ha sido el error más frecuente. En este caso, ese código debería haberlo generado el comando (llamando al repo adecuado previamente) y luego pasarlo en el constructor de la clase del modelo.

Mapeador

Clases con defectos de mapeo

Anotaciones mal establecidas, sobre todo en extremos de asociaciones, tipos de fechas y enumerados. Algunas pasan desapercibidas y generan errores de ejecución.

persistence.xml mal configurado

Por ejemplo. con la línea de (re)generación de tablas sin comentar, lo que lleva a base de datos vacía cada vez que se arranca la aplicación.

Mala gestión de transacciones y/o contexto de persistencia

Aunque es muy improbable que te confundas con esto, casi lo tienes que hacer a propósito, es un error que en un comando se cree directamente un *EntityManager* y se gestione la transacción explícita. Al estar ejecutándose el comando ya desde el *CommandExecutor* la gestión de las transacciones ya está hecha. Ese proceder abre transacciones en paralelo lo que rompe la atomicidad de la transacción principal y puede dejar el sistema inconsistente. Se ha visto también código similar en fachada de la capa de servicios.

Ejecución

No arranca, *orm.xml* con errores en queries o en mapeo

Errores de mapeo o consultas en *orm.xml* que no coinciden en atributos con las clases del modelo. Al arrancar el programa, el mapeador analiza las clases persistentes y compila todas las queries y si hay alguna mal lanza una excepción y no arranca.
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.

Base de datos corrupta o que no se mapea

La base de datos entregada no coincide con el mapeo que se hace del modelo del dominio.
Se interpreta que ni siquiera se ha probado la aplicación para entregar…

Incompleta, le faltan casos de uso 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"…

Revienta con excepciones al ejecutar

Revienta y muestra en pantalla una traza de excepción en situaciones como por ejemplo:

- Al teclear códigos que no existen.
- Al generar certificados que ya están generados
- Al borrar elementos que no pueden ser borrados
- Al liquidar un contrato ya extinto, etc.

Todos son errores de programación y probablemente por insuficiente validación en las clases comando.

Sin datos suficientes para pruebas

La base de datos se entrega vacía o sin datos necesarios para poder ejecutar los casos de uso asignados lo que impide probar la aplicación. O el persistence.xml está mal configurado y cada vez que arranca el programa borra todas las tablas.

Funciona con errores

Aunque se ha implementado toda la funcionalidad pedida y no lanza excepciones hay errores lógicos. Algunos ejemplos son: generar varias veces el mismo certificado, no calcular bien los importes, etc.