JDBC Delivery Checklist

This checklist is not a rubric or an evaluation contract with which you can accurately determine the grade of your delivery. It is simply a list of the most frequent errors in these installments. The intention is that you use it before uploading your project to campus.

This document accompanies the Excel sheet that you must fill out and attach to the submission of your work. Don't forget to hand it in completed.

The error checks discussed below are some of the most frequent detected over several courses, but they are not all. Not having any of these does not guarantee that your delivery will be perfect, but it will be much better than if you do not check it.

Preconditions
Service layer interfaces have not been modified Tests provided by teachers compile Tests provided by teachers are executed correctly These three preconditions must be met so that delivery is not discarded. Interfaces must be respected as they have been delivered. They constitute a contract between the customer and the service provider. Note that a script will copy the original interfaces and tests into your delivery and then run the tests. Therefore, the modifications you have made to your tests and interfaces will be lost and compilation and/or execution errors will be generated, which will invalidate the delivery. Before making your submission, verify this, lest you have inadvertently introduced some change in the code of the tests or interfaces.
Documentation
No readme.txt file In this file you must specify your name, the use cases that have been assigned to you and any other comments that you consider. Not delivering it is penalized. The extended domain model was not delivered in the corresponding task and on the date indicated.
Code
With warnings Your code to deliver cannot have warnings. A warning is a compiler's alert of possible errors. Delivering code with them generates insecurity in the recipient and conveys the feeling of sloppy and unkempt code. All warnings can and should be avoided, either by changing the code in a way that avoids the potential source of error or, in very twisted cases, by silencing it with @SupressWarnings (use the latter responsibly). Java Code Conventions recommendations are not followed The naming, indentation, etc., agreements are important for code sharing in a community of programmers. They also give indications of how to write the code in a way that its structure is understood quickly. All languages have their own. In Java, the "Java Code Conventions" are followed as a basic reference. The statement expressly mentions that the code must conform to this agreement. For the evaluation of this point, attention will be paid to the names of attributes, methods and variables, the use of the camelCase and the correct indentation of the code. A common source of naming problems is forcing the Java attribute name (camelCase for compound names) to match the name of a column in the table (where _ is often used for compound and uppercase names). Lines of code that are too long

The JCC recommends that you do not exceed column 80. Although the limit may seem low to you, there are reasons for it.
If you configure the IDE, it can solve it moderately well when you tell it to format the code, but you have to tell it. And it's not perfect. For example, if you have put a particular formatting in the comments it will remove it and the "fluent interfaces" style is usually not respected. The way to avoid problems is to respect the margins of the code.
This penalty will not be applied for a few lines that exceed 3 or 4 characters of the limit, but it will be applied when it is evident that this recommendation is being ignored.

Incorrect class names (plurals, tables, etc.)

Class names are in the singular, since a class represents a concept. On the contrary, the names of the tables are usually in the plural, since they are interpreted as "all those <things> that are kept there". It is also considered a mistake to name classes as tables in order to avoid having to adapt the mapping with more annotations.

Poorly indented code

Code indentation is important to grasp the structure of the code at a glance.
Examples like this do not allow for that perception:

```
    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;
    }

    }
```

A special case are the parts of code in which the style of fluid interfaces[1] is used.
Maintain vertical alignment.
This is okay:

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

This is wrong:

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

Non-empty methods with comments //TODO

They give the feeling of sloppy code. Clean code is left in a delivery.

Methods with commented code

They give the feeling of sloppy code. Clean code is left in a delivery.

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

Some important bugs

Such as the presence of non-private fields in the classrooms.

Some minor bugs

Small programming errors due to misuse of the language or the usual idioms.

catch NullPointerException

It is considered malpractice. An exception of this type indicates a programming error, that is, if it occurs it is because our code has bugs. What you need to do is fix the code so that it doesn't produce it, but don't deal with that kind of problem.

catch Exception

It is considered another bad practice. Catching *an Exception* treats all possible types of errors that may appear equally. At the very least, a distinction should be made between business logic errors (perhaps the user's fault) and system/programming errors. Generic treatment must be different. Often some stop the program, others do not.

Business

Logically misplaced service façade

The mission of the service layer façade is to hide the classes behind it, but not to execute business logic. It simply distributes, or directs to the appropriate class, the responsibility for solving the corresponding service. If logic is added, it loses cohesiveness (basic design principle) and becomes totum revolutum with possibly hundreds of lines of code and becomes difficult to maintain.

Poorly implemented logic

If you pass the tests, it is difficult for this to happen, but it is possible.

Transaction scripts do not implement the Command interface

All of our service (or business) objects are implemented following the Transaction Script (TS) and Command patterns. For the second it is necessary to implement the provided Command interface.

Commands are not executed using the JdbcCommandExecutor

Commands create connections and/or transactions instead of the executor

If this happens, check the previous point. They are closely related.

Commands throw SQL, Persistence, or Runtime exceptions

Commands should only throw BusinessException or IllegalArgumentExceptions.

Business objects (TS´s) have dependencies on the presentation layer or other business objects

In layered architecture, only dependencies on the layer immediately below are allowed. Therefore, check that there are no dependencies on a TS on ui packages or other service layer objects. Remember that our transaction scripts are implemented in command packages so you should not have import statements from other commands, facades or service factory.

Command Prints in Console

A command is executed at the service layer. Only the presentation layer can display information to the user. To do that in a command is to have misunderstood the overall design of the application.

If you have put *System.out.println()* to make traces or checks while you were developing remember to remove them, when you deliver we will not be able to know with what intention you put it. It will be more difficult for you to forget them if instead of *System.out* you print in *System.err*.

Some commands do not conform to the Javadoc of the interface

The Javadoc of the interfaces specifies the expected results and exceptions to throw.

This error will be considered if all the checks that are specified are not being verified or the logic is not resolved as indicated, even though the operation works in some scenarios.
I advise you that, as you implement the business code, you always have the service interface in front of you. It describes in broad strokes the logic that should be implemented in the service layer and how.

catch BusinessException in the wrong command

This error is considered when a BusinessException is caught in a command code and treated incorrectly. For example: e.printStackTrace() is muted, e.printStackTrace() is printed on the screen, or an error is printed on the console.

TS´s make changes from Record objects to Dto objects unnecessarily

The information obtained from the persistence layer reaches the TS in the form of Record objects (Dto of the persistence layer). The service layer can work with that information. Creating a Dto object is only necessary in order to return information to the presentation layer.

Command objects do not use the persistence factory to use gateways

To decouple the service and persistence layers, the factory pattern is applied, which consists of hiding the instantiation of gateways in a factory. Instead, instantiating those instances on business objects is a mistake.

Persistence

Gateway methods with business logic

The only mission of the persistence implementation is to take care of bringing objects from the database. It is not up to them to analyze that data, validate them, or question the results of the queries, or generate unique codes. They are simply pure and simple CRUD operations.

Persistence Class Methods Throw BusinessException

Usually with methods like the following one:

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


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

The persistence layer does not have enough knowledge to decide that the result of a query, or its absence, is a business logic error. Whether it is a business logic error will be decided in the layer above. And if it cannot be decided there either, it will be in the presentation layer (perhaps it is the user's fault, if he has typed a code that does not exist...).
This point is related to "Gateway methods with business logic", but this time the emphasis is on the type of exceptions that are being thrown.
BusinessExceptions are only thrown by the application.service package.

Persistence Class Methods Throw SQLException Instead of PersistenceException

In order to eliminate the dependency on the database, the Gateway will throw a generic PersistenceException and not SQLException. A method in the persistence layer would only throw PersistenceException (unchecked), which would indicate some kind of system or programming error: referential integrity errors, database connection problems, syntax or semantic errors in queries, etc. In any case, system errors or bugs.
PersistenceException in unchecked as it inherits from RuntimeException.

Gateway objects create and release connections and/or transactions

The management of transactions (and therefore of connections) is carried out in the transaction scripts of the service layer. In this way, multiple Gateways can share the same connection and run in the same transaction.

Gateway objects do not release the resources they create

Specifically, the Statement or PreparedStatement and the ResultSet that they have created during their execution.

Complicated SQL queries

There are queries that implement business logic, such as generating or receiving an order, calculating the amount of an invoice in a select, etc…

Not enough queries

It indicates that data filtering is done in Java that could be done more easily with a SQL query. Instead of executing a specific query, the command makes a call to the "findAll()" method and then programmatically filters. That means a loss of performance and excessive memory usage. The following example serves as a sample:

```
@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;
}
```

Another way to fall for this error is to make findBy methods... on very complex gateways (with Java logic). Instead of making a refined query, a "thick" query is made and in Java, in the gateway method, filtering is done.

Inconsistent queries

Queries that don't return the objects they're supposed to return. Queries on gateways that return objects from other tables and not from the table associated with the Gateway.

There are queries or connection parameters not externalized to the property files

They must all be there, check that you don't sneak some of them into a gateway.

Presentation

There is business logic in Action classes

Another manifestation that architecture has not been understood. The presentation is ONLY concerned with user interaction. Point. If the supplier, the spare part, the supply, the supplier, the supply, etc., are brought to the presentation layer, the responsibility is transferred to the presentation layer. In this case, the presentation would ask for the ID of the supplier, the ID of the spare part and the new price. The logic layer will take care of the rest.

Moving logic to the presentation layer presents several problems:

- Packet cohesiveness is broken (there is a leakage of functionality from business to presentation) and coupling is increased.
- More data is transferred between layers than necessary. It can become critical when the presentation layer and the logic layer are running on separate machines. This is not the case, at the moment…
- Each call to the service layer is executed in a separate transaction. If several modifications need to be made, each of them will be made separately and there will be no possibility of rollback() in case of errors. Extra code will have to be added to expressly undo all intermediate modifications.
- If another presentation layer had to be added (e.g., web in addition to desktop), the same business logic would have to be rewritten. Copy and paste or replicate. This would lead to maintenance problems, etc…

The display layer does not use the service factory to use the facades

To decouple the service and presentation layers, the factory pattern is applied, which consists of hiding the instantiation of service facades in a factory. Instead, instantiating those instances in the presentation objects (action) is a mistake.

catch BusinessException en Action

It is not considered a serious error, but it is a redundancy. With the implementation of BaseMenu, these exceptions are already caught for the purpose of displaying a message on the screen. This is already done by the BaseMenu class in its main loop.

catch PersistenceException o Exception en Action

Out of place. Catching that exception explicitly involves coupling the presentation layer with the persistence layer, which should be hidden from it behind the service layer. A PersistenceException indicates a system or programming error in the persistence layer that is not caused by the user or business logic. The user probably won't be able to do anything. The program terminates and emits an error message.
Catching that exception for the purpose of displaying a message on the screen and allowing the program to continue, indicates that you have not understood how exceptions are handled in the program.

Execution

App won't start

It has compilation errors, it has been delivered with an incorrect version of Java (the one on lab computers must be used, Java 17), etc.
It indicates that the application has not even been tested to be delivered, or that it is delivered knowingly ("let's see if it sticks"), causing others to waste the time that has not been dedicated to the subject.

Incomplete, missing use cases, or doing nothing

A deployment has been delivered that is either missing or incomplete with the commands that implement the use cases.

Bursts with exceptions when running

Pop and display an exception trace on the screen in situations such as:

- When typing codes that do not exist.
- When generating orders for parts that are already pending on other orders
- When deleting items that cannot be deleted.

All are programming errors and probably due to insufficient validation in the TS...

Works with bugs

Although all the requested functionality has been implemented and does not throw exceptions, there are logical errors. Some examples are: generating several orders for the same piece under stock, not calculating prices correctly (bulk errors, not small decimals), etc.