ALEXANDRIA

*Design document*

\*The information in this document concerns the design decisions of the *Alexandria* database project, such as the use given to JDBC or JPA, the approach to XML or the advantages of using a GUI.

**POJOS:**

First of all, we started this project creating the pojos of every entity in the E-R diagram. Each pojo consists of the name of the class (entity), its attributes, constructors, getters, setters, and equals and hashcode methods which have to be override.

In this case we will have a pojo for author, another for paper, disease, body part, device, symptom, procedure and image.

**Then, we created the tables in JDBC:**

To do this, a connection is created in a class called “DBManager”, which is its only attribute and has to be initialized to null to avoid any further problem. This is going to connect the user interface with the actual data base.

When the connection is done, the tables are created only once. At the end of the program the DBmanager has to be disconnected.

The way of programming this is by using CRUD methods (this acronym stands for Create, Read, Update and Delete) which are the main functions that the database should do.

* For the **create** part we have to make a method that creates the schema of the tables with a SQL sentence inside java code. First, the connection is made creating a statement in the connection, then a separated String is written with the SQL sentence that is going to be sent to the statement. After that, the statement has to be always executed and finally, closed. All of this is done inside a try-catch clause to make sure the errors are captured.

The creation of the tables is only done once right after the connection is made. There is a way to manage that, because maybe the user does not remember whether he has created the tables or not. To solve this situation in that method, before creating the tables again, a “Select” is made and if something comes out of the database, the tables are already created and there is no need to do that again.

Also, we will have to create the tables needed to manage the “many to many” relationships making a table with two columns, which are foreign keys, both of them being part of a primary key.

After creating the tables, we have to fill them up with information. That is done with an INSERT in SQL. As done in the create tables, there has to be a method that is going to receive an entire object, being these ones the pojos (author, paper, disease...).

Then, the statement is created with the connection and passed to a Statement object that is going to be executed with the SQL sentence written after that.

The sentence will be of the form: “INSERT INTO nameOfTable (attribute1, attribute2…) VALUES (‘ “ + nameObject.get().... + “ ‘)”;

* The R stands for “**read**”. This is done with a “SELECT” in SQL language.

In the end, the main aim of the database is to store information that the user is going to access and then manipulate. With the selects the database will always search inside itself and it is going to get everything the user needs, working as a filter.

This is done with a sql sentence like: “SELECT \* FROM nameOfTable WHERE id/name LIKE \_\_”

These conditions have to be specified by the user because the select can retrieve everything from the table that they have chosen or it can only retrieve some records from the table that can be searched by id (which is the primary key and it is always unique) or by name.

This is later printed in the console with another method that will only be in the user interface and it is called “show”.

This “show” method is also used to let the user sees the items every time they want to delete, update or select something from the database. The method is going to ask for either the id or if the user wants to see all the list of items. This way the user does not have to learn all the ids by heart, also they are not repeated ever even if an item is deleted from the data base (the PKs are unique and also doing autoincrement every time a new item is created). It is also going to be managed by the data base as it wants.

* The U stands for “**Update**”. In the user interface it is seen as “modify”. In this methods some attributes of the object are sent. This is only because sometimes updating the whole object may be nonsense. For example, an author is not going to change neither their name nor their origin and, of course, the id is not going to change. So, the method Update for the author’s pojo is only going to update its association and, therefore, the method is only going to receive a String that is going to replace the old information.

Here we are going to use prepared Statements, in the deletes, updates and inserts in n-n relationships. There are some advantages of using the preparedStatements.

First of all, the security. They are safer than the statements because the SQL sentences have some ‘?’ that are going to be filled in after loading the statement with the SQL sentence which is done in the creation time. Then, the update is going to be executed.

Also, as it has ‘?,?,?...’ it gives the option to change only selected information. That is why we used prepared Statement in the updates.

In the case of the statement that we use in the inserts (command createStatement), we only have the option to write all of the information of the record.

In the delete part we are going to take care of the “deletes” which SQL sentence is like this: "DELETE FROM nameOfTable WHERE id=?”. Here we use a prepared Statement too.

**JPA:**

There is a little part of the project that we did in JPA. But we must say first that there are some differences between JPA and JDBC. The former is easier to program but it is slower than JDBC. The latter uses SQL queries more than actual objects in JAVA so it is faster when compiling.

JPA MANAGER:

First, to be able to use JPA it is necessary to change some code in the META-INF folder which has the persistence.xml file.

* We have to change the “provider-name”.
* The classes involved. The pojos have to be inserted between <class> and </class> tags, these are going to be tables, later on.
* URL of the project

We created an independent JPA manager so that our project works with both, JDBC and JPA. To do that:

First, an EntityManager has to be created (similar to the connection in JDBC), the command is “Persistence.createEntityManagerFactory (“provider-name”).createEntityManager()”.

Then, the transaction has to begin and the use of foreign keys with JPA has to be activated with “em.createNativeQuery(“PRAGMA foreign\_keys=ON”).executeUpdate()”.

At last, the transaction has to be commited.

In JPA just before and after we want to do any action as insert, delete or update, the transaction has to begin and then commmited.

The way to insert objects into the table is as simple as using the setters of the pojos like: “em.persist(symptom)”.

Second, to read something in JPA the method that is going to read a disease has to get the disease ID and retrieve the whole disease by asking the database through a SLQ sentence that is executed as a NativeQuery to which we have to send the class we want to work on as a parameter of the query.

The deletes do not need to retrieve anything but they need the object id to know which item to delete from the database. To do this the object is going to be read with a “read” method that is going to make a select of that item with its ID and it is going to put that in a variable which is going to be removed from the table like it was a list with the command “em.remove(variable)”. Then, the transaction is commited, as always.

The Update is going to smash the object that is going to be changed with the new one that we have and is going to be sent to the method. And if there is some kind of relation between the object that is changed and another class, the change has to be made in both sides.

This method is not going to retrieve anything and it needs the new description of the object (in this case we are working with procedures and the id of the object that is going to be changed. The object is going to be read with the help of its ID and a “set” is going to be used to insert the new information into the table.

JPA is also used here to insert information in the many to many relation tables. The method gets the id of both items that are going to be linked, read those with the “read” method that is going to make the “select” in SLQ and then, both are added with the “add” method as: “item1.addItem2(item2)”.

In every “one to many” relation there has to be a list of objects in the part of the “many”. In the case of the many to many relations, there is a list in both sides.

For JPA to be used, some things are compulsory to be written in the class:

* The class has to implement Serializable
* It needs a parameter-less constructor
* Getter and setter methods for all the attributes
* Equals and hashcode have to be override.
* Bidirectional associations.
* String toString method and “add” and “remove” methods are optional but recommended.
* Each POJO in JPA has to be annotated, that means writing words like ‘@Entity, @Table, @Id, @GeneratedValue…’ because this is what is going to tell JAVA that the things done are tables for a database. The relations are marked with ‘@OneToOne, @OneToMany, @ManyToOne, @ManyToMany’

**JAXB:**

JAXB is the way of programming the database to later be able to create an XML file. This is important because we want to show the user the information of the database without the need of using an application for it. The XML file helps creating an HTML file that can be used by the user and helps him or her see this information included in the database. For this, some annotations need to be added to the POJOS, like we did before for JPA. These annotations are: @XmlAccessorType(XmlAccessType.FIELD), @XmlRootElement, @XmlAttribute, @XmlElement, @XmlTransient (this one tells the computer, the JAVA attribute is not going to appear in the XML file), @XmlElementWrapper.

The main aim of JAXB is to be able to convert POJOS into XML and viceversa. These processes are called marshal and unmarshal.

For the **marshal**, which is converting JAVA code into an XML file, a JAXBcontext has to be originated. Here is where we are going to create the marshaller that is going to call the “marshal” method and send an Object of a POJO and a file that has been already created.

In the case of the unmarshal it is done the other way around. The same context is used and the **unmarshaller** is created as type Unmarshaller. At last, the unmarshaller is going to call the unmarshal method which is going to ask for a file and it is going to retrieve a cast of the POJO. This information is put in an Object of that same type.

**XML:**

After the JAXB part is done, we are able to create an XML file. This is a file with text organized using tags that structure the content of the project. For this file, a ‘library’ has been created made up of 5 articles that talk about different stuff like diseases or devices. Two different tags are used to create the file: element tags and attribute tags. The first ones are non-self-closing tags, this means they are written like: ‘<tag> elementName </closesTag>’. The second ones, attribute tags, are self-closing tags, these are written like: ‘<tag attributeName = “attributeValue” />’. This differentiation between tags is made by typing with the JAXB annotations ‘@XmlAttribute’ or ‘@XmlElement’.

**DTD:**

This file is important to let other programming users understand the XML file without the need to have a complete look to it. To tell things are attributes or elements, two words are used:element and attlist.

**XSLT:**

After the XML file, this XSLT file is needed to be able to create the HTML file so the user has a userfriendly experience. This file also uses tags, is similar to XML typing, but it determines the way the information appears in the HTML. In our case, we have created a table with the paper’s information. This file is similar to a ‘normal’ JAVA class as it uses things like ‘if-else’, ‘for-each’…

**UML:**

The UML diagram is composed of the classes that are involved in the projects. It is a graphic way to see the components of the JAVA project at a glance.

It is composed by packages (such as the POJO package and the JDBC and JPA manager packages) and their classes. This classes are represented by boxes in which the first part is the name of the class, the attributes come next and in the third part the methods are written without body. In the case of the pojos, the last part will be empty as the pojos do not have any methods apart from the constructors, getters and setters.

Also, the associations and multiplicity between classes and interfaces have to be represented in this diagram with an arrow or a diamond in the case of the aggregation and composition relationships.

**GUI**:

The last part of our project consists of a *graphical user interface* that the user can use instead of the console. It works the same way as the command-line interface but it allows the user to see every item of a class the whole time and the interaction with the database is much simpler.

It is composed of a table where all the items of one type are displayed, buttons to select which type of item is to be seen and three special buttons for adding, modifying or deleting an element. If the user chooses to do so, special windows will be opened to interact with the user. Options to marshal and unmarshall and to generate a HTML are also presented as buttons.

In addition, the user also has the option to view all the related items to one he has clicked on. This information is shown in the lower part of the interface.