Design document: OpsourceSimpleApp

Architecture overview

The application is structured using the **MVC** pattern. The view layer contains basically the MainApp class that reads the user input and prints the information using the standard output. The model contains the class Server, which is a representation of a server description. The Controller contains the logic of the application.

The **operations** are implemented using the **Command pattern**. The main reason to do that is that all of them are very independent and decoupled. Other good reason to do it is that, in the future, this will allow the programmer to implement the *undo* operation. Furthermore, all the operations are served using a **Factory** approach. This factory is in charge to create the different Objects depending on the user's input. The factory is unique in all the package, as it is implemented following the **Singleton** pattern. The decision behind this is because this class uses expensive resources (connections to DDBB or files in the filesystem) so it guarantees that these resources are not duplicated each time we need an operation. The method *getOperation(String op)* is basically a refactor of the switch provided in the main class.

Finally, the persistence layer is implemented using the **DAO** pattern. My approach here was to implement 2 subclasses: to access DDBB and to access XML. It could look like the XML is pretty empty, as it only implements one method, but I think this is the best solution as the architecture is very clean and, in the future, it is a good base in case the developer wants to extend it with more functionality.

Implementation details

The parameters needed to connect to the DDBB has been extracted to an external file. There is no error handling with this file as I have supposed it is always correct (and because the lack of time). It needs to have 4 lines: the JDBC driver string, the DDBB URL to connect, the user and the password.

When the developer doesn't need the DAOs anymore (for example when the application is about to end), it is necessary to free resources. This is handled by the factory class, which creates these instances as well. It implements a method to call the *closeConnection()* in each one of the DAOs.

The xml parsing is done in an inner class inside the XML DAO. It has been implemented using annotations from the *java.xml.bind* library. It has been the only part where I had to research on Internet about how to do it.

To handle the errors in the application, I have used some *Exception* classes from the Java standard library like *UnsupportedOperationException* or the *SQLException*.

Testing

I spent around 5 hours to finish the exercise, so I didn't create a very exhaustive test battery. I tested the most delicate classes which are the DAOs (DDBB and XML) using **unit testing**. I tested each one of the methods with limit values and strange operations to force bad behaviours.

I created an **integration test** as well working directly with the DAO objects and simulating how the interface calls the methods. I tested not only the expected behaviour, but the non logical situations as, for instance, try to delete a server which doesn't exist.

Future steps and improvements

In the future, the architecture is open to be extended in order to support new functionality. For example, in case **new operations** are needed, it would be only needed to include a new branch on the *switch* statement in the Factory class, apart from the new class extending from *Operation*.

If a new **persistence laye**r is needed, it is just as simple as create a new class implementing the ServerDAO interface. It will automatically ask the developer to implement that methods.

One thing that could be good to improve is how the **operations receive and display information**. They actually use the standard input and output, but this breaks the MVC pattern. This tasks should be done in the view. I didn't implement it to simplify the exercise, but the good implementation is to set a List as unique parameter in the *execute()* method, so they could receive the input from the user in that list and then return the output to the view.

Finally, in order to improve the tests, the DAO classes should **not access to real DDBB and files**. They should simulate this to simplify the tests and avoid errors not related with the implementation.