Software Architectures: Assignment 2

Architectural Patterns

Pieter Meiresone, Lars Van Holsbeeke

March 2015

1 Introduction

This report corresponds to the 2nd assigment concerning the course *Software Architectures*.
It contains findings and solutions to major design flaws coming with the assignemt, with extra paid to the layered structure of the application. In the ideal case, one should be able to change any layer without having to change an other.

2 Weakening the coupling between application and database layer

2.1 Required Functionality

Making it easier to switch between different types of databases (SQL relational and JSON), one should create as less entrypoints in the database layer as possible. At the moment, `DatabaseFacade` is the main entrypoint of the datasbase-layer. However, if we take a closer look at this class, we can see that it creates three types of databases:

- `UserDatabase`: the database subject to user management and the only one considered in this report.
- `RegularDatabase`
- `RawDatabase`

Looking to the methods implemented by the `UserDatabase` class, one can observe the `insert(UserProfile profile)` and `update(UserProfile profile)` methods, invoking both the `executeSql` method getting `profile.asSql()` and `profile.asSqlUpdate()` as parameter respectively. Both `asSql()` and `asSqlUpdate()` methods return SQL-code as a string but are implemented in, depending on the type of user profile, `FreeSubscription`, `CheapSubscription` and `ExpensiveSubscription`. Though these classes belong to the application layer, they generate pure SQL

code which depends on the type of database one uses (in this case, an SQL database).

Conclusion: the coupling between the application and database layer is, based on the previous, to tight and should be weakened.

The current goal is to remove the `asSql()` and `asSqlUpdate()` methods from the application layer and move them to the database layer, which does (and should) depend on the type of database (SQL, JSON,...) being used.
First, we create an interface for all three databases: `UserDatabase`, `RegularDatabase` and `RawDatabase`. Doing this enables us to use different implementations for each of these databases. In our case, only *userDb* is concerned.
Second, inserting (when creating new users) and updating (logging last login time when user logs out) *free*, *cheap* and *expensive* subscriptions should be supported. SQL code has already been provided for each subtype of the userprofile class. This code can easily be reused by creating three methods for both inserting and updating userprofiles in the `DatabaseFacade` and `UserDatabaseInterface` (the interface created out of `UserDatabase`).

- inserting a new userprofile
  - `insertFree`
  - `insertCheap`
  - `insertExpensive`
- updating a userprofile
  - `updateFree`
  - `updateCheap`
  - `updateExpensive`

Depending on the type of database being used, the implementation of these methods differs: in case one wants to use p.e. a JSON-style database, the class providing its abstractions should implement all of `UserDatabaseInterface`s methods. Let us continue with the SQL case, in which `UserDatabase` implementes `UserDatabaseInterface`s methods. As already mentioned, in the old case two methods were used: `insert` and `update` calling the `asSql()` and `asSqlUpdate` methods of `UserProfile` respectively. These now have been replaced by the six methods described previously, having the SQL insertion/updating code directly in their body, decoupling the application layer from the type of database being used. Not to forget: one still has to provide getters to the private fields of a userprofile that have to be kept in the database, otherwise the databaselayer can't access the data it needs to save.

Until now, only changes at the side of the databaselayer have been discussed. Yet, modifications also need to be done at the side of the applicationlayer. As

already mentioned, there is no more typechecking (free, cheap or expensive subscription userprofile) invoked by the databaselayers. Hence, this has to be done at applicationlevel in one way or another. A possible solution to this, is creating an abstract method `insertToDatabase` in the superclass: `UserProfile`. Each of the subclasses (free, cheap or expensive subscription) then has to implement this method by invoking the corresponding method of the databasefacade (insert/update free/cheap/expensive).

2.2 Other design flaws

In general, the coupling between the database and the applicationlayer is to tight. This is not only the case for subclasses of `RegularUser` (which have been fixed in the previous section), but also for the subclasses of `Administrator`, a subclass of `UserProfile` too. These classes also generate their SQL representation (methods *asSql* and *asSqlUpdate*) on the applicationlayer, making the whole implementation dependent of an SQL-based database at the databaselayer. The same approach being used for the subclasses of `RegularUser` can be applied to fix this problem and reduce the coupling.

An identical problem can also be found in the other classes of the datapackage (`Article, Book, Conference` ...) on the application layer: they also contain their own SQL representation, making the whole implementation again depending on an underlying SQL database. To decouple here, a similar approach can be used by adding insertion,... methods in the databasefacade. Depending on which kind of underlying database one wants to use, he has to implement the methods of the databasefacade for that kind of database.

Looking once again to all the classes of the datapackage, one can observe they all have, besides `asSql()`, also a method to serialize the class to XMLformat: `asXml()`. This method is used to represent objects (implementing this method) front-end. The way it is currently implemented assumes that only xml will be used, coupling the application to xml technology when communicating with the client. A possible solution to overcome this flaw, is creating an interface declaring all methods needed to serialize an object to a certain front-end representation and having it implemented by any technology one wants to use: xml, json, ...

When having a closer look to the coupling between the UI (*InternetFrontEnd*) and the application (*DatabaseFacade*) layers, we can find a good example completely violating the seperation of layers concept: *ApplicationFacade.getRawData()*. This method returns 'raw' data directly from the databaselayer. It is called at UI level `AdministrationPage.doGet()`. This proves that a part of the application logic is embedded in the UI layer, preventing the development of a new UI layer without having to duplicate the logic.

General conclusion: looking to figure 1 in the assignment, one would ex-

pect that all communication between layers passes only through the dedicated interfaces (`IApplicationFacade` and `IDatabaseFacade`). This is often not the case. As a consequence it becomes way more difficult to use other technologies together with the existing system in the future. Our advise: use the interfaces as provided by the figure. In this way the whole program can be decoupled from a specific implementation, making it easier to change layers.

3 Databaselayer

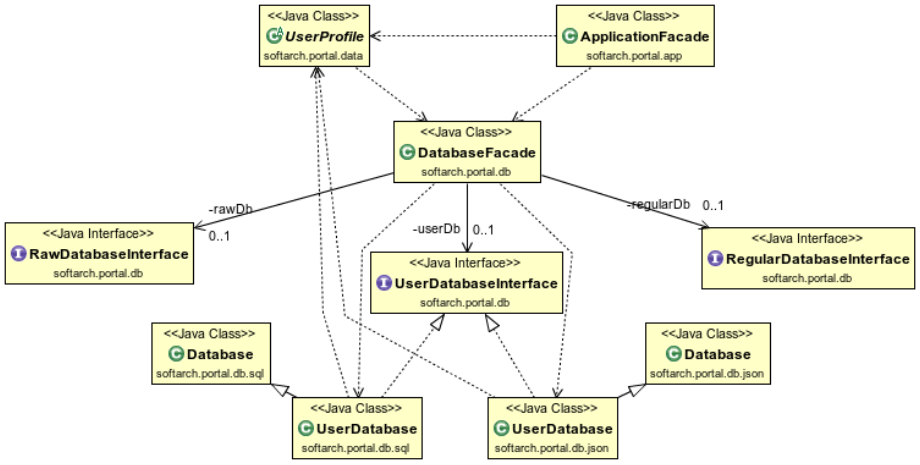

Figure 1: Database layer classdiagram

4 Switching between the implementations of a layer.

In the configuration file web_portal.cfg an extra field is stored to indicate which database layer should be used. The field “dbType” can have values “sql” or “json”. The field “dbUrl” should be changed accordingly.

In the constructor of the “DatabaseFacade” an extra parameter is added, indicating which type of database should be used, depending on the parameter classes from the package db.sql or db.json are instantiated.

Thus the facade pattern allows us to easily switch between implementations of a layer. Since each layer is accessible to other layers through the facade (the facade defines an entry point to a layer), we can easily switch between different implementations of this layer in the facade.

References

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.