Cross-reviewing – Project Team 7

In this document we have put together different aspect of software development, as they were reflected in the ESE project code of team 7. To make this easier to follow, the first part it is split into 4 sections: Design, Code Style, Documentation, Tests.

Questions and comments are always welcome.

Regards, Team 4.

Part 1

First Section: Design

- Violation of MVC pattern

It seems that the Model, View and Controller are usually well separated and therefore the MVC pattern has been well used. There are some cases, though, in which classes seem not to appear in the right place. For example the class *PictureManager* is located within the controller classes. It would seem more appropriate to associate it with the services. From Spring MVC tutorial: "*A Controller is typically responsible for preparing a model Map with data and selecting a view name*". *PictureManager* does neither. The calss *NominatimConnector* seems to have the same problem. Another example is the *EnquiryComparator* and *EnquiryComparatorRating* which are included under Models. This might not be so much a violation of the pattern as just bad location for the classes which results in the code being less easy to understand. We will therefore note this again under Code Style.

What could be considered a violation of MVC pattern is the excess of logical functions in search.jsp. As this is a "view", it would seem appropriate to transfer some of the logic to a Controller and Services (especially, for example, the definitions of maximum and minimum prices for an apartment, which are important factors in the placement of an ad as well as in a search for an ad, yet they are only defined and appear in this view).

In our opinion the responsibilities in the different layers (MVC) are defined very well.

- Usage of helper objects between view and model

With the use of different forms ("pojos") as helper object the view and model are interacting well. They are well implemented and are properly used.

- Rich OO domain model

Most domain concepts that play a role in the application and assume responsibilities are objects in the software. An example for a model which is implemented but not used is Address. The results in a very big Ad model (see comments under Code Style).

It is worth mentioning here that we have found it interesting to discover that an Ad object is containing a Picture object as well as every Picture object is containing an Ad object. This makes the dependencies between the models confusing and we think this could (and should) have been avoided.

- Clear responsibilities

As far as we could see, each class has clear responsibilities and not more than one main responsibility. We have found the big number of classes (especially notable among the controllers) to be very effective in the understanding of the code.

- Sound invariants

Unfortunately we have not found invariants.

- Overall code organization & reuse, e.g. views

Over all, we have found the general design of the code good. There are many classes with clear responsibilities and it is obvious that thought has been put into the connection between the different components of the system.

Second Section: Code Style

- Methods

When it comes to code style in the method level, the code is mostly well written. All through the program we have found the methods small in size and effective, which makes the understanding of the system much easier.

- Naming

The names of classes, methods and objects have been most of the time cleverly chosen to reveal their use, with some notable exceptions. In the *SearchConroller* class, for example, the *SearchDao* object is given the name *searchRepository* (similar in *AdDao*). This is especially confusing since the class is relatively long and therefore it is difficult to maintain an overview and follow the flow. A standard *SearchDao searchDao* would make this much easier. Another example for problematic naming is found in the Strings "us" and "you" in *AdForm*. Those names are very confusing as it is unclear who are they referring to. We have also found the name of the class *AuthentificationController* rather confusing. A different name, such as *LoginConroller*, would reveal the reasonability of the controller better, just like the rest of the view-controller dependencies, which seem to carry the same name. We would like though to mention again that those examples might be especially obvious because the names have been otherwise mostly well chosen.

- Keeping classes and methods in the right place

The project, though not huge, is still big enough to make it not easy to follow the flow of commands, and therefore a good organization of the classes is important. Unfortunately, this is not always the case, and we will give a few examples. As mentioned in the Design section, some classes are not located in the project in the place where we

assume they suppose to be. This creates confusion and makes code inspection time consuming. The examples of the classes *PictureManager* and *EnquiryComparator* have been already mentioned. It also seemed to us that part of the search logic is located in the class *NotificationServiceImpl.* It would probably make more sense to put such logic operations in a search service, so it could be found easier when needed. We have also found the service classes not very clearly organized which resulted in difficulties following the code. We will give one example: The retrieving of ads using an AdDao is done in *AdServiceImpl*, in *EnquiryServiceImpl*, in *BookmarkServiceImplas* well as in *NotificationServiceImpl.* Again, we think the code will be much more readable if it would be organized in such a way that the name of the service will demonstrate what kind of service it provides.

- code repetitions

As mentioned already in the first section, an Address model have been created with attributes like street, city, zip etc., but it seemed not to be in use while the Ad model has the same attributes. The result is unnecessary repetition and "inflation" of the code. The intention might have been to use an Address inside an Ad, but this has not been done. More examples can be found in the services. As we already mention before, the service-classes names are not consistence with the services they provide regarding the retrieving data from the data base, and this sometimes results in code repetition. One example is the call *adDao.findOne()*, which is found in *BookmarkServiceImplas, AdServiceImpl* and in *NotificationServiceImpl*. This could probably be avoided by reusing an object or creating a dedicated object with this responsibility.

- Exceptions, assertions and class invariants

It seems there is a lot more which will have to be done in this area, especially regarding assertions and invariants. Exceptions seems to be handled in some places, where there are mostly needed (login, registration etc.).

Third Section: Documentation

Where documentations are present they are understandable and mostly well written, although some small aspects could still be improved, for example the use of plural first person "we" in the *Software Requirements Specification* document. We believe that a passive form would be more appropriate.

The effectiveness of the comments in the code, which are short but exact (for example, a very good comment in *NotificationController*: *"Creates a model displaying user's notifications*") reveals precisely the intention of the code segment as well as describes the responsibility of that segment.

The biggest problem we encounter regarding documentation, though, was the absence of it in many places where we thought would be very helpful. None of the models, for example, have a class comment. Obviously, this is not always necessary. Everybody could assume what an Ad is, but the difference between an Inquiry and a Search, for example, is not clear. Furthermore, some classes which are located in the models section are not models, and there is no explanation, for example, of what *EnquiryComparatorRating* does. Similarly, class comments of the services would be in some cases very useful.

One issue with the existing documentation is that it is not always consistent with the domain vocabulary. We will bring one example from the *Software Requirements Specification* document: in scenario 8 we read "auto delete" but in the alternative scenario we find "the ad goes inactive". It is not clear what happens with the ad and what exactly the difference between deleting in becoming inactive is.

Regarding use cases, we thought more details would make it easier to understand and then to implement. For example "Searcher and Provider schedule appointment" seems much too general and does not specify what exactly happens. Another thing which is missing in the use cases are pre- and post-conditions. In scenario 8, for example, we only find one pre-condition and one post-condition. There are some important conditions which in our opinion are missing, for example "the system gives a warning one week before end date", "the system deletes the ad by the end date" and so on.

Fourth Section: Tests

We have found the tests clear and distinct. They are easy to read and it is very easy to see what is tested thanks to correct and understandable naming and design of the test classes.

The creation of test data could be improved and become more efficient. One example is in *LoginServiceTest*, where the same data for the *signupForm* is created for both tests. This could be avoided by using the @Before method or with the help of help-methods.

The main drawback is that the coverage of the test cases is rather small. Also, for some reason many of the tests did not pass on our computers, but this might be due to conflicts in data base configurations. Extending the testing to other parts of the code would seem appropriate and would have to result in extension of the test data.

Part 2

The class we chose to examine and analyze is the *SearchController* class. This controller contains 5 methods, all of them bearing a *@RequestMapping* annotation. Two out of these methods are responsible for performing the search process while the remaining three handle saved searches. The number of methods seems reasonable and all methods are connected with different views which are representatives of different aspect of the search function of the application, and therefore in our opinion correctly placed.

The method “search”, However, not only controls the */search* order by using service-objects and provides the correctly equipped Views, but performs the search itself. We believe that in order to maintain the basic principles of OO design as well as to comply with the MVC pattern, passing the *searchForm* to a method in *SearchService* which will perform the logical part of the search process would be more appropriate. The actual search is clearly a service-task, and therefore should be uncoupled from the controller. The methods *saveSearch*, *searches* and *removeSearch*, on the other hand, focus on mapping and generating the correct view, while it is mostly service-classes who perform the actual modifications, as expected by controller-methods.

Another aspect of the code which we found problematic was the use of numbers instead of variables (“magic numbers”). This makes the method more difficult to read and understand. The use of constants like DEFUALT_roomSizeMax would immediately explain their purpose and thereby would make the code more readable.