Project Review Room4you

By Team 8

Reviewers:
Michael Grünig
Sara Peeters
Daniel Ziltener

November 17, 2014

Contents

1 Design

1.1 Violation of MVC pattern

The MVC pattern is not violated in the design.

Classes of the controller are contained in the `ch.room4you.controller` package. The model is found in the `ch.room4you.entity` package. Connection between the model and the database takes place through the data acces objects (DAOs) in the `ch.room4you.repository` package. All interactions between the controller and the model are caried out by the services in the package `ch.room4you.service`.

There is however a difference between the interpretation of the MVC pattern by this project group and the interpretation presented in the exercise hours of ESE. Being that in this project, model entities are used directly as backing objects for the forms, instead of heaving extra form pojo's in the controller taking this role. Strictly taken this can be seen as a MVC pattern violation, since model objects are directly used in views. The extensive usages of form pojo's as backing objects however leads to lots of duplicated code. This is a clear example of a design trade-off.

We do understand the design decision of team 3. Especially since they do almost all validation on the client side using javascript . Only checking whether a user is unique is done serverside, since a database query is needed for this task. This eliminates the main responsibility of the form pojo's. (An exception is the user entity, where all checkin is done in java again. We could not find a reason for this inconsistency)

The only place where a form pojo is used is for the search form. This is because a search is not a model entity. The question can be asked whether it could be useful to have searches also represented as entities that are persisted to the database. Using this information, statistics could be made that might be interesting for the administrator and/or the custommer.

1.2 Usage of helper objects between view and model

View requests on the server are handled by the different controllers of the `ch.room4you.controller` package. The interaction between the controllers and the model takes place through different service objects. This is implemented in such a way that the controllers job is limited to calling the right method of the right service, and the actual processing is done by the services.We can conlude that the interaction between view and model is well designed.

1.3 Rich OO domain model

There is an object oriented domain model, but calling it rich would probably go to far. The main shortcomming is that the design does not reflect that there are some essential differences between different kind of ads, as well as a lot of similarities. Each ad for example has a title and probably pictures, an ad-placer etc. However a shared apartment will focus on roommates, which rooms are shared, which languages are spoken, what shared activities take place etc. whereas this is not of any importance for a normal apartment. As the website evolves maybe other ad types will be added, and doing right to the differences and similarities of different kind of ads by using abstract classes and inheritance will improve flexibility in such a case.

1.4 Clear responsibilities

Although class responsibilities are poorly documented, the actual distribution of responsibilities is appropriatly done.

Several controller classes as well as several service classes corresponding to there respective model entities have each the responsibility of handling request regarding these model entities.

One remark that could be made is that some services are coupled to upto 5 DAOs. So the well distributed responsibilities go to the cost of a high coupling.

1.5 Sound invariants

No invariants were explicitly defined. It would certainly be useful that the team would give a thought to what the minimal valid state of the entity classes is.

Because the entities have only getters and setters and are usually externally manipulated, a kind of public invariant checking that none of the obligatory elements stays empty could be useful, even though it is a bit unconventional.

1.6 Overall code organization and reuse

In Java itself the code organization is close to excellent. Some room for improvement stays in the JSPs.

For example code could be shared between the new ad form in `user-account.jsp` and `editAd.jsp`.

Another overall organizational remark is that in this design an alert subscription and a search are looked at as 2 totally different things, even though actually they aren't all that different.

2 Coding style

2.1 Consistency

2.2 Intention-revealing names

There are no cryptical names, neither for classes, methods, nor for local variables.

2.3 Do not repeat yourself

2.4 Exception, testing null values

2.5 Encapsulation

2.6 Assertion, contracts, invariant checks

2.7 Utility methods

3 Documentation

3.1 Understandable

The documentation in the SRS is well understandable and structured, though incomplete and lacking e.g. what is part of a user profile.

The same goes for the code documentation: The javadocs are, when they exist, well understandable, but mostly trivial and incomplete.

3.2 Intention-revealing

The SRS tells us about the intentions behind the use cases. The javadocs lack that information, though.

There are no class-level javadocs, and almost all method javadocs are completely trivial, just describe what the method does but lack description of what the method actually is for.

3.3 Describe responsibilities

Responsibilities aren't described anywhere in the javadoc strings, nor in the SRS.

3.4 Match a consistent domain vocabulary

4 Tests

4.1 Clear and distinct test cases

There are 8 test classes, distributed over two groups(each in their own package), controller tests, and service tests. There are also two testsuites, each with a different number of tests. Although some tests appear in both testsuites, none of the testsuites contains all 8 tests. Moreover, we note that some of the tests in the service test package seem tot be testing data access objects directly and not services.

Although the idea behind the organisation of the tests is good, the implementation lacks consistency and is confusing. The fact that no javadoc is provided for most of the tests doesn't help either.

4.2 Number/coverage of test cases

As said two groups of classes are said to be tested: controller and services. There is no explicit mention of any tests testing the data accessing objects of the repository package. However the UserTest actually is a DAO test. There is not even a userservice mentioned here! (Even when assessing this test as a DAO test it is still insatisfactory, since it doesn't test whether an id has been set to the entity after it has been saved to the database.)

The other servicetests are also not such a great contibution to the overall coverage, since most of them seem to test a mock(eg. `findAdsMatchingToAlertCriteria()`and `saveAlert()` from the AlertTest class, `saveAd()` from the AdTest class), or a get method from a list(eg. `findAdByUser()` from the AdTest class), or nothing at all(eg. `createNewAlertTest()` from the AlertTest class), but not a method of the service it is supposed to test.

Controller tests are very basic and certainly don't cover the majority of the controller cases, but at least they are real.

We have to conclude that although lots of tests have been writen, no service is actually tested. The controller class is basically covered, an the UserRepository is the only DAO that is, even though not correctly, tested.

4.3 Easy to understand the case that is tested

Given the above it is hard to believe that the creators of the tests(at least those in the service package) do themselves understand the cases that are (not) tested.

4.4 well crafted set of test data

Except for one case, the password in the UserTest, which doesn't represent a plausible password, since it wouldn't get through the validation, the test data are reasonably well crafted. Even though not a lot of special cases are tested(no input etc.)on places where user input is expected.

4.5 readability

Most tests are not ducumented, which does not necessarily have to be a problem. However, not all names of test methods are meaningfully chosen. The alertTest class for example has a test that is simply called `test()`, as well as a test `mail()`.

5 controller class evaluation: UserController

The UserController class seems to be responsible for all the requests coming from the my account page. Well, on a second look, not all the things, but most of them. The problem with this is, that

most of these things have to do with ads, and not with the user, as one would expect from the name. Also, things that one would expect here, such as changing user profile, do not appear in this class.

It is especially strange that there is a mapping here for `/ad/removeBookmarkAd/{id}`, whereas a very similar method appears in the AdController for the mapping `/ad/unBookmarkAd/{id}`. One of them is probably at the wrong place. On the one hand since the BookmarkAd part is also in the AdController, I would be tempted to move it there, but on the other hand the method makes use of the UserService, which would again plead for the UserController.

It turns out that what seems a clear responsibility at the beginning, in the end is to scattered since it goes about actual users things like deleting the account, but also about ads, bookmarks and alerts. It is clear that this class can be improved, either by moving some methods to the AdController, making a BookmarkController, or making the name more meaningful. The optimal case in my opinion, would be to keep in this class only stuff that has really to do with the user, by moving other stuff to other controllers that might still need to be made (BookmarkController, UserAdController...).