Project Review Room4you

By Team 8

Reviewers:
Michael Grünig
Sara Peeters
Daniel Ziltener

November 14, 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.

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

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

3.2 Intention-revealing

3.3 Describe responsibilities

3.4 Match a consistent domain vocabulary

4 Tests

4.1 Clear and distinct test cases

4.2 Number/coverage of test cases

4.3 Easy to understand the case that is tested

4.4 well crafted set of test data

4.5 readability

5 controller class evaluation