Cross-Review for Team 6

Design:

The code follows a clear design by using the MVC pattern. The code is separated in model, view and controller. The model represents the data by entity-objects, DAOs, and POJO-helper objects. The view is realized with jsp-views and the controller by the controller classes and the inherent services. The classes have clear and definite responsibilities.

Suggested improvements:

- As the controllers and services use the POJO-helper-objects, they could be moved from the “model”- to the “controller”-package.
- In our opinion, @Autowired annotated objects don’t need constructors. Therefore, the constructors in the controller classes could be removed.

Coding style

The coding style is consistent and free of repetition. The naming is meaningful and intention revealing. Own exception classes are created and the variables are encapsulated. There is a considerable number of null value tests.

Suggested improvements:

- Use more assertions, contracts and invariant checks.
  For example, invariant checks could be indicated for the quite complex method of AdServiceImpl class.
- Some methods are very long and complicated (e.g. the equals method of Address class [see figure 1], both the show and the save method of AdvertController class, as well as the saveFrom method in AdServiceImpl class [see figure 2]). Utility methods could simplify the code (e.g. createAd or createAddress helper methods).

```java
@Override
public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Address other = (Address) obj;
    if (adr_id == null)
    {
        if (other.adr_id != null)
            return false;
    } else if (!adr_id.equals(other.adr_id))
        return false;
    if (city == null)
    {
        if (other.city != null)
            return false;
    } else if (!city.equals(other.city))
        return false;
    if (coordinate == null)
    {
        if (other.coordinate != null)
            return false;
    } else if (!coordinate.equals(other.coordinate))
        return false;
    if (plz == null)
    {
        if (other.plz != null)
            return false;
    } else if (!plz.equals(other.plz))
        return false;
    if (street == null)
    {
        if (other.street != null)
            return false;
    } else if (!street.equals(other.street))
        return false;
    return true;
}
```

Figure 1: equals-method of Address class

Figure 2: save-from-method of AdServiceImpl class

Documentation

All classes except the interfaces have at least some documentation. For reusability purposes it would be helpful if also the interface had some kind of documentation, so that the programer knows what he can expect form the interface.
The entities have little documentation, which in our opinion ist not a big problem, because the getter and setter methods are self explanatory.
The best documented classes are the controllers which makes sense since they are the core of the application.

For the most part the documentation is understandable. Nevertheless it contains some typos and grammar errors which some times make it hard to understand the intention of the function and should therefore be corrected. Furthermore the abbreviation adverts is used for the advertisements which does not match the naming convention with the service in which the abbreviation “ad” is used. The names should be changed accordingly.

The biggest lack in the documentation is that every entity contains a function called equals which is never explained and also not very straightforward. To see the intention of the function one has to read carefully trough a lot of if statements.

Every function documentation states the parameters which have to be provided to the function and the return which can be expected. This gives a deeper understanding of the function. However sometimes the parameter description is a bit confusing. In our opinion the names of the parameters should be chosen so that no further description is needed. This would simplify and clarify the function documentation.

All in all we would state that the documentation is comprehensive and helps to understand the code to a certain degree. Even where the documentation contains some ambiguity, it is still better than no documentation.

Tests

1. Clear and distinct test cases

In general the test cases are clear and specific. They are also well structured, short and cover only one main Functionality. Yet the naming of some test cases could be more meaningful. For example: testShow() in class AdvertControllerTest.java.
It is not obvious, what should be shown exactly. Show adTitle or what?

2. Number/coverage of test case

There exist 4 packages with 16 test classes altogether. From the 71 created test cases, 70 pass and 1 method fails the test. (public void toString() in class UserTest). There is a coverage of 71.0% for the Skeleton. This seems to be quite good at the moment.

3. Easy to understand the case that is tested

The test cases are easy to understand with few exceptions. For example testSave() in class AdvertController. The assertion in this test is, that when the method model.getModel().get("title") is called, the return value should be “Access Denied”. Either the naming for this test is wrong or there is an error in the test case’s logic.

4. Well crafted Test Data

In general, the test data is well chosen. Boundaries were checked and mock objects were used. Unfortunately, there is no test in class RegisterController which assures that the InvalidUserException is thrown when the contract is violated. Furthermore, there are as well

test cases which we think are unnecessary, because they check third-party function. As for example whether the generated id is greater than 0. The test cases cover mostly basic functionality.

5. Readability

As already mentioned the naming for the test cases could be improved. Although there are not any comments or documentation, the readability is okay. The whole organization could be improved if the tests are structured in a TestSuite.

LoginController

The controller contains a get request mapping for /login and /logout. In our opinion the error handling in this controller is complicated for no need. The developers built an extra class which generates errors or success messages. With the HTTP protocol and the mvc pattern the messages could be passed as request parameters in the url itself. The intended message or error could be passed via request url and then be caught and handled directly in the view.
Other than that the controller is structured very well. The paths /login and /logout are exactly what you can expect from a login controller. However only the get request for the /login url is in the controller. For reusability purposes it would be good if the post request was also in the controller, so the programer could control the program flow after a successful login attempt. In the current state only an erroneous login attempt is handled by the login controller.