Review of ESE Project of team 8

From: Team 5

Date: November 19, 2014

For the coding style and design review we skipped the security part, since it’s only a copy of the framework Mozilla Persona and not relevant for this task.

Design

The MVC pattern was not violated which means there is a clear distinction between Models, Views and Controllers so that the code will be easily maintainable. . The helper objects so called pojos, between view and model, are used correctly. But there exist two objects (*ProfileForm* and *NewProfileForm*) that do exactly the same and hold the same code. You should really avoid that, it makes the maintenance a lot more difficult because you don’t know which one is used for what.. A very good approach is the splitting of RealEstate, SharedApartment and Apartment, it makes the database access much faster and provides more agile maintenance.
Just a little remark, you should place the Class *PersonaSignupHandler* in the folder service, since it is a helper class which uses interactions with the database and therefore it’s not a Spring MVC model. Also it’s a little critical to use database access within an object. Normally you would select the user from the service and then put it into the Object via Constructor parameter passing.

The classesin general have clear responsibilities. The methods and variables have names, which are self-explaining To understand the code much faster it would be recommended to give the saveFrom methods different names.
There is only one invariant called *checkDates*, try to use more invariants to make debugging easier. Use the assert function for this in the beginning of your methods. In general you should first check if every necessary value is available and then start with what you actually intended to do.

Overall you could think about splitting your IndexController. Because at one point it will become huge. Which also means, that merging can get chaotic if several people are working on it. And from pure design view you directly see what the controller is doing, especially when you give them fitting names like UserController, RegisterController and so on. A good approach is to split it by functionality. You use a good indentation and markup style what makes the code easily readable but it’s not done everywhere. Try to make it more consistent and check for false formatting before release v2.0.

You’ve extracted two JSP views to prevent code redundancy. This is pretty neat, keep on like this. But you have duplicated and nonsense views like *showAd* vs. *viewAd*, *index* and *main*. Make sure that you don’t use synonym words in the future. Also you could dynamically show and hide profile information when a user is logged in by the use of <c:if test> and Javascript.

But in general the code has a good design and has only a small amount of repetition. It is just not possible to run your code in this version.
You probably use another version of java. You should tell the reviewer next time how to get your code running.

Coding style

The file naming is mostly good but there are some incosistencies which make the understanding of the intention difficult. For example, you’re using two or three written out words but then there are two files using a number: *Team8Authority.java* and *Ad2.jsp*. It’s not very clear what these numbers say and so it’s even less clear what the files do. The last filename which is unclear is *ShApartment*, this is an inconsistency due to the use of not written out words and because a file exists named *sharedApartmentForm.jsp*.

There are a few non intention-revealing names for class variables, e.g. *fixedMoveIn* and *fixedMoveOut*. An added “Date” would help a lot, and to mark it as a boolean you could add a ‘b’ in front, like bFixedMoveOutDate. The same thing with the actual date variable.
All other model objects are fine. So let’s continue with the pojos. Here you use twice the absolute same code for almost the same purpose: *ProfileForm.java* and *NewProfileForm.java.* You could have used only one which is reusable. It’s even quite unclear where in jsp you use *NewProfileForm.java*. Normally you would create a pojo form class only when you want to pass data from jsp to java. *SignupForm.java* then triples the repetition of the register class, a little search has shown that it is not used in the project (probably debris of the plain Skeleton project). The exceptions are fine, but there should be one or two more, at least one for each form.

Now continuing with the *SampleService.java* and its implementation. The basic markup is good and the methods are mostly clear. But there are four methods which are redundant in pairs. *saveFrom(ApartmentForm) → saveFrom(Apartment)* and *saveFrom(ShApartmentForm) → saveFrom(ShApartment).* This should be avoided because it causes confusion. There is also a method *saveFrom(signupForm)* which again is debris from the plain Skeleton. The basic markup is quite consistent except for the method *loadUserByEmail()*. You should always use the same style. Some methods use the @Transactional annotation and some don’t, e.g. you could place this annotation to the whole class *SampleServiceImpl.java*, since almost every method uses a DAO. The same with @Override on Line 75, this is not needed because the method has to be implemented here and there’s no superclass.

A very good approach is the use of helper methods *checkDates()* and *isFutureDate(),* they fit smoothly into your code. But you could also add some more checker methods, e.g. that a field cannot be empty and so on. You do check for empty dates but not for other null values. This may cause NullpointerExceptions.
In general the encapsulation is maximized, all setters and getters are set to public and all variables private or protected, but there is almost no javadoc available (only in indexController).

Documentation

SRS

Generally, the SRS is detailed and represents the app pretty much. The language you use is fine..
The Use Cases meet the Use Case diagram. There are 14 Use Cases which are described in detail and always in the same schema (actors, description, trigger, preconditions, …). For every Use Case the preconditions and postconditions are set so you can easier see the responsibilities from each of them. Yet, there are some points to improve: For faster reading it is recommended to use the same person for making sentences. For example the 'Description' sometimes begins with 'As a user I want to…' and sometimes with 'The users want to'. Also it is easier to read if the naming of the actors is consistent. As an example you can take a look at Use Case no. 13. The description of the actors is 'Ad-placing user, searching user, administrator'. After this definition 'user', 'administrator', 'actor' and 'user or administrator' is used. At this point it is not fully clear what a 'user' or an 'actor' means. It would be better if exactly the predefined terms are used for a whole Use Case and even better if in all UseCases the same terms are used.
All needed roles are described at the end of the SRS. These are users and administrators. It would be helpful if this description would be moved before the Use Cases, so you already know what these roles mean when you start reading the Use Cases.
The SRS contains goals of the system, which is a good idea to support the customer. There is also a section for definitions and abbreviations but it's still empty. Here are some ideas for this section: 'admin', 'DBMS', 'candidate', 'enquiry', 'ad placer'. All these words appear somewhere in the Use Cases and may be described somewhere. But it is helpful to put them into the 'Definitions'.
A small mistake: In the table at the beginning it says that there is only one version on the date October 17th. But the date of the SRS is November 11th and the team has written: ‘The developers commit to updating the document after each customer-meeting, assuring that it represents the project specifications in the best possible way at any time.’

Javadoc

In the following are some chosen classes reviewed in detail to show how Javadoc could be improved.

IndexController.java

The three methods which have to do with searching have Javadoc. The Javadoc is understandable and contains basic information needed (overall description, parameters, return value). Additionally, you could add pre- and postconditions to those methods where it makes sense. This can be done with Javadoc or asserts.
Most of the other methods can be understood with a little effort. But for example 'initBinder(WebDataBinder)' really should have a Javadoc because you don't automatically know what it means to 'init a binder'.

SampleServiceImpl.java

Again, here is only one Javadoc for one method which is sufficient. There are the two methods getApartmentsByUser() and getShApartmentsByUser(). You should describe what is meant by 'Sh' or at least use a word that is self-explanatory. Maybe 'Sh' stands for 'Shared', so the method would be getSharedApartmentsByUser(). It doesn't matter if a method name is rather long or short. Instead, it does matter when a method name is not self-explanatory. You have the same problem with your classes and variables.
Within the method saveFrom() there are some comments. After the statement 'user = userDao.save(user);' there is the comment '// save object to DB'. Because we know that anyObjectDao.save() saves any object to the database this comment is not needed. Just after this line there are two lines of code which are commented out:

```
    // Iterable<Address> addresses = addDao.findAll();  // find all
    // Address anAddress = addDao.findOne((long)3); // find by ID
```

When publishing the final version there shouldn't be such lines anymore. But of course you can leave those in an unpublished private version for developers.

ShApartment.java

This class' head says 'public class ShApartment extends RealEstate { …'.
Neither for this class nor for the class RealEstate exists Javadoc. Even in the SRS there is nothing about a 'sh apartment' or 'real estate'. Because these terms are never used it is necessary to add Javadoc for clarification.

SearchForm.java

The Javadoc for this class says: 'The searchForm at the moment only contains very basic search criteria.'
It is perfect that it is said something about the content of the class. Since you know that this is the Javadoc for the class SearchForm you could just write 'At the moment only contains very basic search criteria.'

Maybe you could also summarize this basic search criteria for example with 'At the moment only contains very basic search criteria, i.e. zip code and categories.'

UserDetailsService.java, Team8Authority.java, and more
The author and creation date of the class is in the Javadoc but nothing else. As soon as not all members of the team add this information to their Javadoc it is not too useful. For this classes, at least an overall description needs to be added to Javadoc.

search.jsp
Some code lines are commented out. As in java files it is recommended to delete these lines for the final version at the end of the semester.

Test

There are mainly two test cases. Checking saved apartments and exception tests. There are a lot more methods to test.You should test other Daos for example. In the Dao tests some use cases are missing. If you test a Dao you should also test what happend with invalid input. For example what happend with an zipcode with only 3 digits or with letters? Try to cover as many different cases as possible.

So, going into the test in detail your Apartmenttest ist practically the same as the SharedApartmenttest. Therefore you should connect them. Better would be a Before section as you did in your SampleServiceTest. There you could initialize address and all other variables and then check for them in different cases. In different methods you can then check if both, normal apartment and shared apartment dao, works.
Your test cases are easy understandable, but you also just test 2 short things (Street and zipcode). The address consists of 3 parts, so you could initialize an address and with the use of assertEquals(apartment.getAddress(),address); you can check for correct implementation.

In your last SampleServiceTest you mock the apartment daos, but it’s not so clear why you do so. If you write a short comment why mocking is useful here would be very helpful. It is good that you also test exceptions but you should check all of them.
But it’s good that you just test one thing per method.

Analysis of Controller class: IndexController

Your *IndexController* class is good. There are all necessary methods which are kept short and easy understandable. It would be good if you comment all methods, a little comment is better than none. Especially when you use uncommon and single used methods/objects like *webDataBinder* or redirectAttributes.
Probably you could merge the two search methods within one. The use of @RequestParam(required=false) would make things easier then.
Since you only use one single controller it has too many responsibilities, you should split it up into at least two new controllers, called UserController (which is responsible for registration, login and profile) and AdController (which is responsible for creating, editing, showing ads).

The method makeAd is called from two forms at once. Don't do that, this is really bad design. Either each JSP form should call one method or you could merge both forms into one. Basically the error must have been made when you started to distinguish between *SharedApartment* and *Apartment*. The best approach would be to have split JSP views for both types. To create actual links to the site it would have been possible to use a prefix for each. Let's say all apartments start with an offset of 10'000 or so, while all *SharedApartments* start with 0.