

Design



Violation of MVC pattern:

I couldn't find parts in the code, where the MVC pattern was really violated. There are some parts where the views control not only how the data is displayed, but also what data is displayed. e.g.

```

```

However, according to me, this doesn't seem to be a problem.

Usage of helper objects between view and model

Due to the clear separation of the models, they used only few helper objects. I see as a helper object the 'Row.java'. This class has only one responsibility and is only used for the updateSubjectForm, so this is not a problem. However it would be nicer to have directly 'subject' entities displayed in a datatable, because subject already have attributes like 'name' and 'grade'.

Rich OO domain model

Well done. Objects are nice and proper designed. Nothing more to say about that.

Clear responsibilities

Except of the point above (MVC Pattern) the responsibilities are well organized. Model classes are only dataclasses like desired.

Sound invariants

I couldn't find any invariants because from my point of view there is no complicated logic used outside the framework that would require invariants.

Overall code organization & reuse, e.g. views



The code is organized well, outstanding is the big javadoc for every method (e.g. in the controller classes). There are not many parts where code could be reused except for the views. The views are divided into a header and content part, which is a good example for code reuse.

Coding style

Consistency

The codebase is consistent. The classes are grouped into corresponding packages and the projectteam used java conventions like camelCase and lowercase starting method names etc. When different classes provide similar services, the names are given also similar, e.g. saveForm saves a form etc.

Intention-revealing names

Well done. The methods are named intuitively and descriptive. Very nice and easy to read.

Do not repeat yourself

There is one small repetition in the Logincontroller create method

```
        if (!result.hasErrors()) {
            try {
                registerService.saveFrom(signupForm);
                model = new ModelAndView("signupCompleted");

            } catch (InvalidEmailException e) {
                result.rejectValue("email", "", e.getMessage());
                model = new ModelAndView("login");
                model.addObject("loginBoxVisibility", "hidden");
                model.addObject("registerBoxVisibility", "visible");
                model.addObject("registerCancelButtonAction",
                    "window.location.href='/login'");
            }
        } else {
            model = new ModelAndView("login");
            model.addObject("loginBoxVisibility", "hidden");
            model.addObject("registerBoxVisibility", "visible");
            model.addObject("registerCancelButtonAction",
                "window.location.href='/login'");
        }
    }
```

in the other classes I couldn't find much duplicated code.

Exception, testing null values

There is an exception for every service class, the only place where they are needed. In addition, almost every method of the different ...ServiceImpl classes has its nullchecks at the beginning. In the other classes there is no need for testing null values.

Encapsulation

Encapsulation is done according to the spring framework. Like intended by the framework, the data is encapsulated quite well. All fields in the classes are private and can only be modified by getters and setters. In addition, for every data model there is only one controller and only one service that modifies the data. This provides good encapsulation.

Assertion, contracts, invariant checks

Except of the nullchecks I mentioned above, there are no assertions or invariant checks. However there is not very complicated logic, so there is not really need for different assertions or invariant checks.

Utility methods

The use of an availability enum makes sense in the util.availability class. The different availabilities are very well documented directly in the declaration of the enum. I don't see any need for other static methods in the project.

Documentation

Understandable

Due to the intuitive naming of classes and methods the code is easy to read. There are long javadocs for almost every method and class, which helps to understand the methods and functionality. Sometimes, there is almost too much javadoc, e.g. in the ShowProfileController:


```
/**
 * Constructor for testing purposes
 * @param appointmentService
 * @param userService
 * @param profileService
 * @param timetableService
 * @param subjectService
 * @param prepareService
 */
@Autowired
public ShowProfileController(AppointmentService appointmentService,
                             UserService userService, ProfileService profileService,
                             TimetableService timetableService, SubjectService > subjectService,
                             PrepareFormService prepareService)
{
    this.appointmentService = appointmentService;
    this.userService = userService;
    this.profileService = profileService;
    this.timetableService = timetableService;
    this.subjectService = subjectService;
    this.prepareService = prepareService;
}
```

The code is organized very well and very readable formatted. Getters and setters are indicated with small comments, that saves time for the reader.

Intention-revealing

As mentioned above, the names are intention-revealing. So is the whole project. Overall, the separation in different models, controllers and services makes sense. Also the data carried by the different forms and model classes is intuitive and chosen well.

Describe responsibilities

According to the MVC Pattern the responsibilities are separated across the code. The project team didn't violate the MVC pattern, so there are clearly defined responsibilities. In addition, the different functionalities are provided by the right classes and do not overlap. 

Match a consistent domain vocabulary

From my point of view they used consistent domain vocabulary. They used javadoc in a proper way and used the advantages of java in a good manner. E.g. consistent use of interfaces for the services.

Test

Clear and distinct test cases

The tests for the ShowProfileController are done quite well. Every other involved class is mocked, so that this unit test supports really the essential part of a unit test. However the other controller classes have no unit test at all. I assume that's according to the lack of time.

The other tests for the different Serviceimplementations check only the intended class and the specified method. With the help of mocks every other involved class is replaced. Bravo.

Number/coverage of test cases

There are numerous test for the services and they test the important functionalities. However, there could be more unit tests for the controllers. There is an additional integrationtest for the LoginController with sufficient test cases.

Easy to understand the case that is tested

Due to the naming and the clear separation of distinct test cases it is easy to understand which case is tested. In some tests there are even help comments to see what part is really tested, thatâ€™s useful for the understandability and the readability.

Well crafted set of test data

The test data is chosen good. There are even some negative tests with expected errors. For the most classes it would not make sense to check special boundaries because there are not really boundaries.

Readability

As mentioned above: Easy to read due to help comments and good formatting of the different test cases.