IST 412: Implementing Design Patterns | Ethan Balcik

1. Introduction

Recently, each of the members of our group have been tasked to identify and implement an object-oriented design pattern, as well as a user-interface design pattern, within our project. Of the many potential choices for each of our design patterns, I chose to move forward with identifying and implementing the "Chain of Responsibility" object-oriented design pattern, as well as the "Input Feedback" user-interface design pattern. In this readme file, I explain precisely where and how each of these design patterns emerge within our software. Furthermore, click this link for a live demonstration of the design patterns in action.

2. Chain of Responsibility

It turns out that we might argue, within our project, there exist two instances in which the chain of responsibility object-oriented design pattern emerges. The first of which is likely the most prominent example, in which, by virtue of our choice to use the Spring framework's "@RequestMapping" annotations, we see a chain of responsibility emerge. We see this as a result of the Spring framework's use of what may be called a "front controller" or perhaps a "dispatcher". By adding "@RequestMapping" annotations, we are essentially configuring our front controller to map certain requests to the function that it decorates. For example, if we add "@RequestMapping("/home")", any time a request is made to navigate to the "/home" subdirectory, the "@RequestMapping" annotation ensures that the function it decorates will run as a result of the request, and ideally it will handle the request with the expected behavior.

Another instance in which we might argue the chain of responsibility design pattern emerges would be the layered structure of Spring. We have 3 main layers in addition to our view layer on the very front-end, and our models/database on the very back-end. Namely, these layers consist of the controller layer, the service layer, and the repository/persistence layer. We have already looked somewhat in-depth into the controller layer and how it, itself, implements the chain of responsibility design pattern, but we still have yet to discuss the specifics of how the controller layer communicates with the service and repository layers, creating yet another chain of responsibility. In particular, when a function within the controller layer runs as a result of a front end request, it has the option to either handle the request entirely on its own, or, if the request requires some back-end action, it can call methods in the service layer, essentially passing the request on to the next layer to be handled further on the back-end.

As a result, at a high level, we have a conditional structure emerge in which requests are sent, distributed among controller methods, and then potentially passed on to be handled by the service layer and repository layer. This essentially creates the chain of responsibility design pattern.

3. Input Feedback

When interacting with our application, the user should expect some feedback in response to their input. Otherwise, the application will not capture the user's attention, it will be deemed unusable, and it will not be adopted. In our application, we instantiate the input feedback user-interface design pattern to make our application more usable. Namely, when logging in, we have two inputs, and thus, 2^2 possibilities for handling input. First, we have the instance in which the user inputs an incorrect email and a password which does not exist in the database, as well as an incorrect email and a password which does exist in the database. These can be combined into one case due to the fact that we query for users by the provided email, and thus, there can be no situation in which we have an incorrect email, but still find a user with a password matching the one which was input. Therefore, we lump these together as one case in which the user inputs an email which corresponds to no user in our database. Further, we have the case in which the input email is correct, but the input password does not match that which is stored by the user which was returned by our query of the database. Then, finally, we have the successful login in which both the email and password match the user returned by our query of the database.

Also, an additional note worth mentioning, we assume that, for any given email, there exists only one user in our database. The reason we can make such an assumption is because this will be something controlled after we implement our register use case. No user should be able to register for an account using an email which already exists within the database. For the time being, we have inserted records manually into our database to demonstrate our login use case, but in the future, this should be handled exclusively by the register use case.

Anyways, to correspond to these possibilities for login input, we have a few feedback prompts which can be output. Namely, when an incorrect email is input, we output "User with email (input email) does not exist". This is also the scenario which takes place when there is no input whatsoever, or when there is just a password which is input, as there should exist no user in the database with a blank email. Furthermore, when a correct email is input, but the input password does not match that which is stored by the user returned by our query of the database, we output "Incorrect email or password". Finally, when correct user credentials are input, we simply carry out the intended function of the login, and send the user to their home page.

To be specific about where and how this takes place, this is handled by our "loginUser" function within the UserController class. Its implementation consists of a conditional statement which checks for the various conditions related to correct versus incorrect feedback which we've discussed above. It either updates the feedback label underneath the login form on the front end, or it calls the "toHome" function, "logging the user in" to their home page.