TEAM: Michael Brughelli, Chris Flauta, Jing Guo, Stephen Rowell
TITLE: Classroom Scheduler
FINAL REPORT

Final Implementation and Features

The final implementation has changed somewhat from our original design. Although we were able to implement quite a few of the features that we had intended, we were not able to implement a few others. In its current iteration, our system is able to add and remove courses, generate a course list from a text file, view the current schedule, request a change to the schedule, and run the scheduling algorithm. However, we were not able to implement a feature that allows for the creation of new users to the system as well as the ability to reset a password associated with a specific user. This was mostly due to the fact that, once we had gotten into the technical aspects of implementing the code, we realized that these features did not really make sense with respect to the context of of our system and were not worth the time that it would have taken to get them working properly. This is different from our original intent, as is illustrated by our previous class diagram, and has been reflected in our current class diagram shown below. Despite this, however, it was very helpful for us to have made this diagram before actually coding the system. Even though we did not implement every intended feature, we had a very good idea of how certain attributes of the code should relate to other attributes. This gave us an effective roadmap for how our system should look after the actual coding was complete. Also, the schema of the database we implemented assisted us in comprehending how our code would actually work.

The changes in the class diagram were not as extensive functionally as they seem like they would be looking at them side by side (class diagrams are the last two pages of this report). Our original service interface was broken into several specialized classes and we also implemented listener classes for the different tables and the actions they can perform and cause

other tables to perform. Instead of just one main view that changes as you complete different tasks, we went with a few view classes that were called upon after main was already running as needed. We scrapped the difference between the department head and student user types and instead went with just an admin user and a “normal” user that had limited privileges (i.e. not being able to run the scheduler). Doing the first diagrams was a great help to us in getting the system to where it is now. They gave us a good stepping stone for which to base our new ideas off of and they allowed us to see how things could be done more optimally. It also cut down the time it took for us to actually implement the system. If our team had just blindly started coding, we’d still be trying to get the first use case to work and instead, we have a really cool project that works and could easily be built upon.

Design Patterns

While developing this system, we attempted to use several design patterns: observer, adapter, and command. In the case of observer, we set up our interface such that, when an object in the course list is modified, the resulting change would automatically be reflected in the schedule. This saves the user time since each change would not have to be done manually for the every aspect of the system. We also have the ability to filter the schedule and manipulate what is shown to us by department and by a specific class. For adapter, we implemented two different login types: admin and a “normal” user. Depending on which of these users is logged in, the GUI will change to reflect the functionality appropriate for that user’s level of access. This was accomplished by creating a new class that used the button action listener and the view generating classes in a different way from the admin user without compromising the existing functionality of the admin user or the button action listener class and the view generating classes. Finally, for command, we set up a queue in which requests for changes in the

schedule by non-admin users could be stored. Admin users may implement these requests at their discretion.

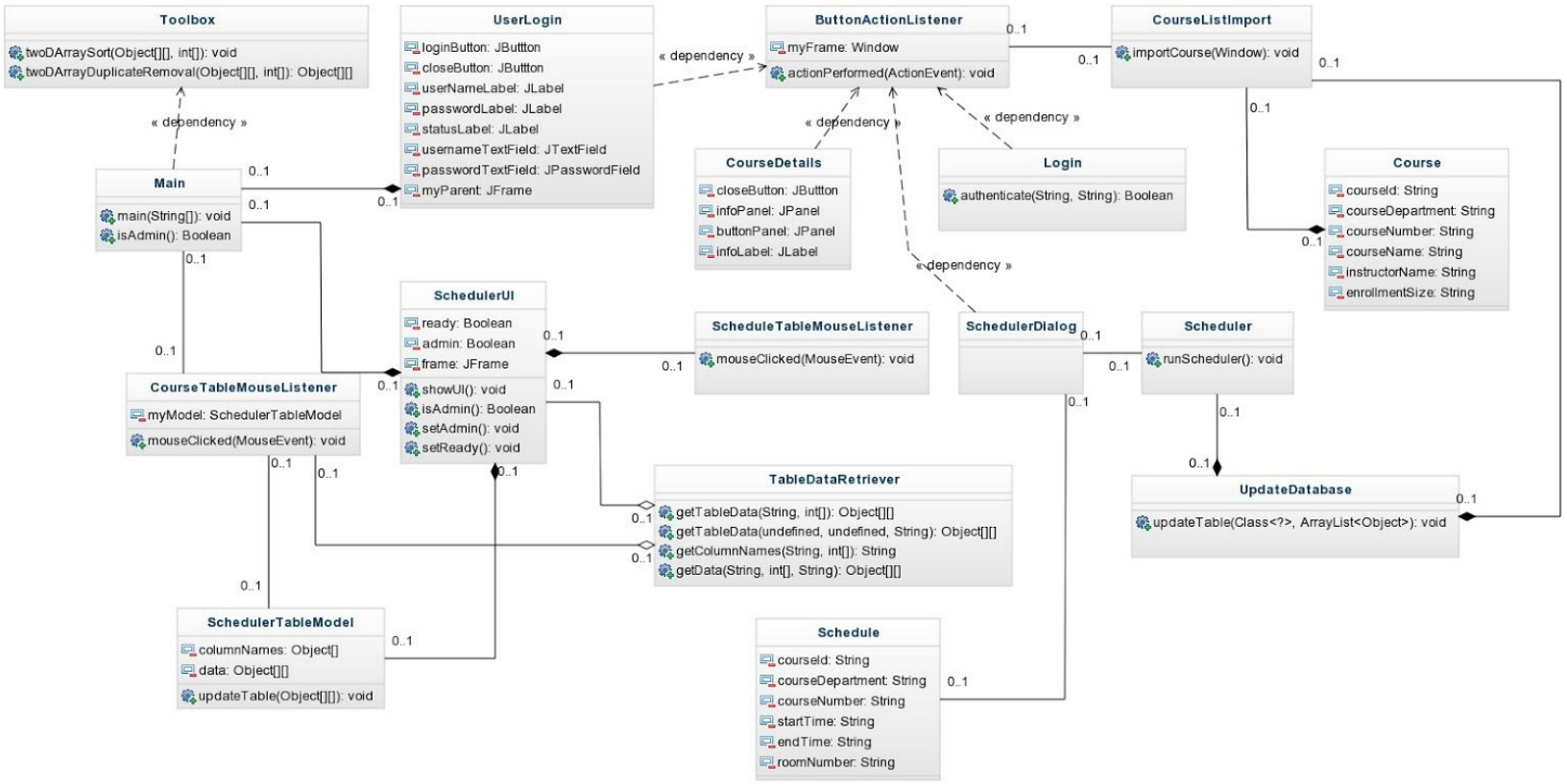
When we came up with our ideas for part 2, we didn't really consider any specific design patterns and we hadn't really started covering them in class too much yet either. But using our basic design and following our requirements and use cases from part 2, we were able to see how to implement the design patterns mentioned above and it made our project implementation go more smoothly and the final product turned out to be better than what our team had originally envisioned.

If you look at the diagrams below, you can see that our original intent was to create a service interface that had several abstract methods for each type of user class - these were going to be functionalities that the users would have differing types of access to. The other methods were actions that they all had access to and functioned the same way across each type of user. We maintained the main view class and added other views as well which we linked to the button action listener class in order to change the views presented and modify the views so that normal users did not have the privileges that the admin users have. We also added a scheduler user interface class that used other classes including a class that acted as an observer. That observer class is what allows our program to filter the data from the schedule and add/ remove classes to the schedule and add/ remove classes and departments from the course list table and department table. The final big change was the addition of the course list importer class which allows us to import class lists to push into the course table in the database with comma delimited files separated into rows by newline characters. This was an added functionality that we thought would be really cool to implement in the last few days of implementing our final code to be presented to the class.

What we have learned

Overall, we have learned a lot about analysis and design for object oriented systems from this project. Previously, none of us had ever stepped through such an intricate design prior to diving into the code. Although this level of design required a significant time investment on our part, it was certainly worth the effort. Since we had planned so many of the fundamental aspects of our system in advance through class diagrams, database schema, use cases, sequence diagrams, and activity diagrams, the process of implementing our code was significantly easier. We also learned how to incorporate design patterns within our project in order to help ensure that the means by which we implemented our system were the most efficient and effective possible. On the other hand, we also learned about what coding practices to avoid in order to prevent anti-patterns from occurring. Despite taking care to avoid those tendencies, we still discovered a possible anti-pattern within our code- poltergeist. This arose in the form of classes that only hold data and have no methods that actually manipulate the data. This made us realize just how easy it is to make these types of mistakes when writing code. If our team had another month or two to fully optimize our project, we'd likely see another overhaul of the class diagram and more functionality as well as improved performance. Undoubtedly, we will take this experience on to our professional careers and mistakes made during the implementation of this project will be avoided in the working world.

FINAL DIAGRAM



ORIGINAL DIAGRAM