Table of Contents

Project Description

The Counseling Center scheduled over 14,000 appointments over this past school year. They currently use a pen and paper method for scheduling appointments for their Initial Appointments (IA) and Emergency Coverage (EC) sessions for UIUC students. Collecting the availability and conflict information from counselors by hand and then using a complicated set of requirements to schedule appointments is a time consuming task that can be improved through the use of technology. An application for collecting availability and scheduling information can allow the Counseling Center to more easily schedule appointments for its counselors and serve its clients more effectively. This application would improve the task of scheduling appointments, and improve the ease with which future appointments can be scheduled by automating the time consuming and difficult task of assigning counselors and clinicians to IA and EC time slots.

Process Followed

For the duration of this project, we followed the Extreme Programming (XP) process. We decided on this software development methodology in the first of our weekly Monday meetings in the basement of Siebel Center. In those meetings we would also plan out and make changes to our project by discussing the status of our user stories and use cases for the current and upcoming iterations. For most of the iterations we split our team up into different pairs and allocated equal amounts of expected work to each pair.

Each pair tended to work on the same laptop as in standard pair programming. While pairs implemented tests along with the user stories, we did not always implement tests before features as specified in test-driven development. For tests, we implemented JUnit tests like last semester, as well as GUI tests using a Swing testing library called UISpec4j. UISpec4j tests are similar to the httpunit tests in that both simulate the effects of clicking and typing on components of the interface. While UISpec4j could not test the effects of dragging and dropping, it worked well for the other GUI elements.

For refactoring, some programming pairs used plugins like FindBugs, PMD, and UCDetector to look for places with questionable code. Others utilized the testing environment and each other to find possible bugs and logic errors in the code. The problems found by the tools did not really match the code smells discussed in lecture, although they did point out some relevant mistakes that we fixed. Many of our refactorings were done in response to new information we obtained from meeting with the Counseling Center. During the earlier iterations, we refactored by moving long and duplicate code into individual methods. In later iterations, we started removing unnecessary code once we were sure the code would not be necessary.

Besides the meetings within our group, we also met with the Counseling Center to clarify project requirements and update them on our progress. During the meetings, we found out that some of the requirements were not what we interpreted from the proposal sent to us. They also added some additional ideas after seeing our demo. Besides discussing functionality, we also talked about how to install and integrate our project with their system after we are finished.

Requirements and Specifications

The formal requirements for our project are described in the document *CounselingCenterProposal.pdf* in the doc folder of our repository. The main tasks are to gather and store information about when clinicians would be unavailable for counseling sessions because of Counseling Center meetings or other commitments. This availability information is then used to generate a schedule that assigns clinicians to emergency coverage (EC) and individual appointment (IA) sessions at predefined times throughout a semester. The interface for inputting that information should resemble the paper forms they currently use. In addition to clinician availability, the schedule must also fulfill other constraints meant to make sure that IA and EC sessions are evenly distributed between the clinicians, that clinicians do not have major commitments both late one afternoon and early the next morning, and that clinicians have time to eat lunch. Besides functional requirements, our project must also integrate well with their system. Specifically, this means our project must be able to store and retrieve information on their *Microsoft SQLServer* database.

The typical workflow of the scheduling application follows the following process. An administrator will revise and update the list of clinicians which are available for a particular semester. They will then enter in the new semester settings establishing start/end dates, holidays and number of hours assigned to each clinician for IA and EC appointments by default. Next, the clinicians will each submit their preferences to the system which includes time they need off and their preferred time slots to work. The admin can then go through and modify clinician preferences or hours assigned to a particular clinician until they are satisfied. Next they generate a schedule which creates IA and EC appointment schedules for the semester. They can freely modify the generated schedules and will be notified if they break any schedule constraints. When they are content with the final schedule they can choose to print or save the schedule to a file. As such, the main components of the system are as follows where each main component consists of several user stories:

A. New Semester Settings
   a. As an admin, I can create a new semester calendar.
   b. As an admin, I can input general meetings per semester

B. Clinician ID List
   a. As an admin, I can add and remove clinicians from the clinician list.
   b. As an admin, I can edit a clinician's availability for a semester.
   c. As an admin, I can constrain feedback when editing a schedule

C. Schedule Generation
   a. As an admin, I can generate appointment slots for a semester
   b. As an admin, I can generate assignments of clinicians to the available appointment slots
   c. As an admin, I can generate a schedule that breaks as few constraints as possible

D. Clinician preferences
   a. As a clinician, I can input my availability for a semester.

E. Schedule modifications and storage
   a. As an admin, I can edit a generated schedule
   b. As an admin, I can save a generated schedule

c. As an admin, I can view and print a generated schedule

F. Other User Stories

a. As an admin, I can validate that the generated scheduled meets the specified constraints
b. As an admin, I can view the history of changes made for a semester
c. As a user, I can set atypical meeting options
d. As an admin, I can edit the database configuration file

Our two main use cases for our application:

Use Case: Generate Schedule
Primary Actor: Admin

Goal in Context: Allow admin to generate semester schedule using clinician preferences and semester parameters

Scope: Scheduling - this action will run an algorithm that assigns clinicians to time slots in the semester that best follows their preferences and avoids time conflicts while following the semester parameters.

Level: User Goal

Stakeholders and Interests:

Admin: Wants to assign a clinician to all available time slots in the semester so that they have more time coverage for their clients

Clinician: Wants to fulfill their responsibilities in their preferred time slot without conflicting with prior meetings/arrangements

Precondition: Semester parameters are set, Clinician ID list is finalized, Clinicians have updated their preference forms

Minimum Guarantees: Able to generate a schedule that assigns clinicians to time slots while minimizing conflicts and maximizing clinician preferences

Trigger:

1. Admin clicks button to generate schedule

2. Admin views schedule.

3. Admin can now save, edit, or print schedule.

Extensions:

1a. If a schedule without conflicts cannot be arranged, still generate a schedule that minimizes the conflicts and show the admin the specific conflicts that prevent creating a conflict-free schedule.

3a. Editing a schedule will prompt the admin if any new conflicts arise after changing something.

Use Case: Clinician Input Form
Primary Actor: Clinician

Goal in Context: Allow clinician to add time away plans, initial appointment/emergency coverage conflicts, and emergency coverage preferences.

Scope: Pre-Scheduling Preparation - this GUI is required for clinicians to add their meeting conflicts and preferences

Level: User Goal

Stakeholders and Interests:

Admin: Needs all clinician input data to be able to generate a schedule for those clinicians

Clinician: Gives ability to influence final schedule to suit their own needs (meeting conflicts) and preferences

Precondition: A semester calendar has been created and selected for clinicians to input their data into

Minimum Guarantees: Able to enter time away plans, IA/EC conflicts, and EC preferences. Able to come back and edit their conflicts and preferences after they have been submitted.

Trigger:

1. Clinician clicks Semester IA/EC Preference Form

2. Clinician adds their weekly conflicts, time away plans, and EC preferences.

3. Clinician clicks update button. Their preference data is saved within the system and will be used by the admin when generating a new schedule.

Extensions:

2a. The dates in time away plans must be within the semester start and end dates and follow time chronology.

2b. The time in IA/EC conflicts must coincide with their respective IA/EC time slots.

3a. When adding a weekly conflict or time away plan, all fields must be filled with their respective input format (time would be a time, day of week would be mon, tues, wed, etc..).

Architecture & Design

The architecture of our system is built around the Model-View-Controller design pattern. This separates the logic associated with the user interface (the View) from the persistent storage logic (the Model), and the complex business logic (the Controller). The View is implemented in Java using a combination of Swing components within a JFrame or a JPanel. The controller is implemented in Java and the model is implemented using a combination of SQL and Java.

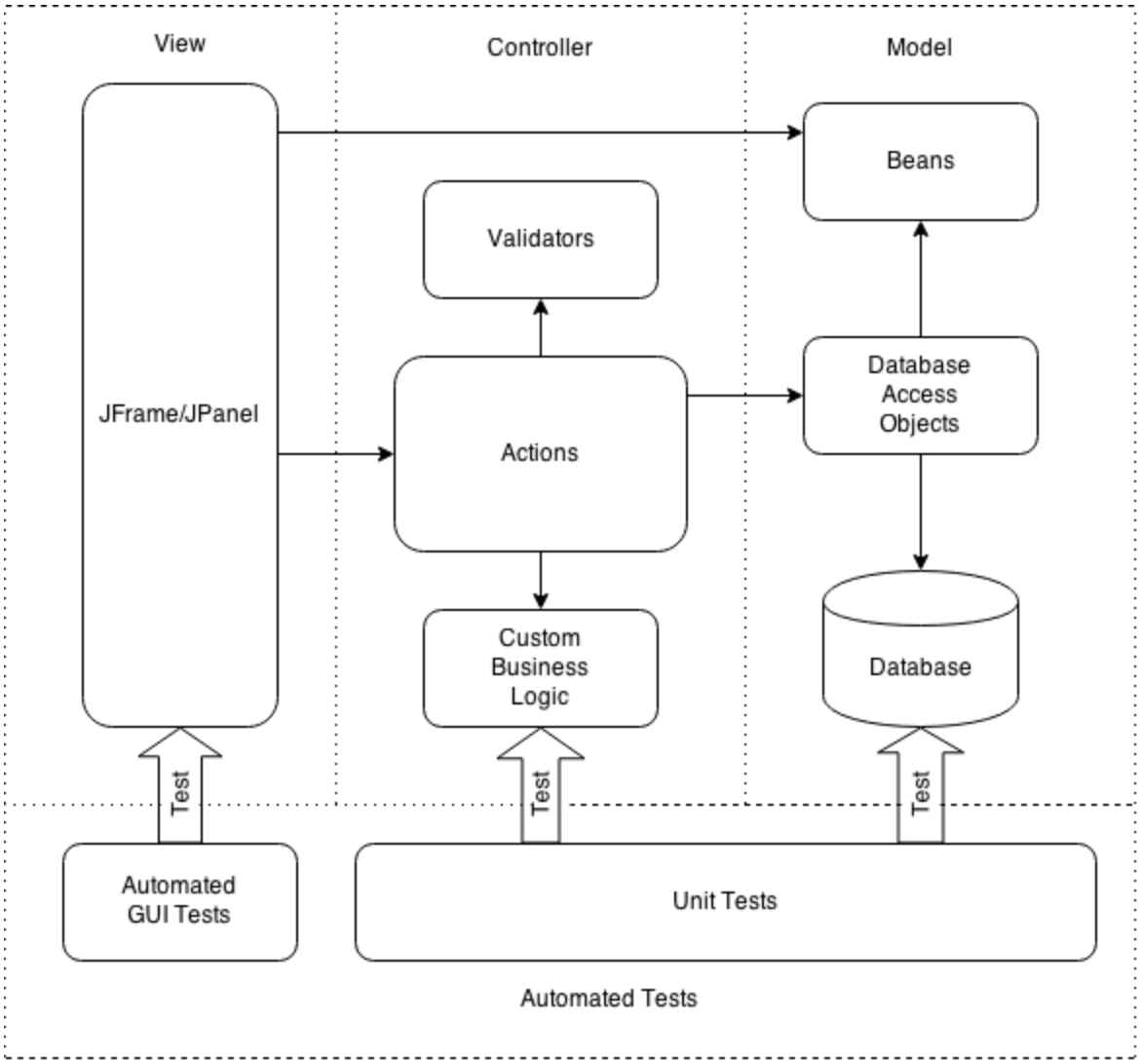

Fig4. Architecture Overview

View

The primary purpose of the JPanels and JFrames is to provide a desktop based GUI. Each JPanel or JFrame contains several JComponents and utilizes different Action classes in order to handle the business logic of the application. The Action classes are instantiated by the JPanels and JFrames which implement the View.

Controller

The overall purpose of the controller in the system is to provide an intermediary between the user interface (GUI) and the persistent storage of the database. It provides the complex business logic so that the View can focus on displaying the information to the user and invoking the controller components in order to provide the functionality of the application.

Overall the tasks the Controller are concerned with mediating between the Model and View, and delegating the logic to the appropriate classes. In our application this involves delegating any complex input validation into a Validator class. Additionally, it delegates the custom business logic to the Action classes which provide the core functionality of the system. The database interactions are performed by the Model, which provides access to the database and persistent storage. The Controller is also utilized to propagate useful exceptions to the View, which can display messages to the user to indicate if an input is formatted improperly or if one of their actions is invalid.

The primary classes in the Controller are the Action classes. They provide the complex business logic involved in the application which involves interactions with the data contained in the model in order to provide the different functionalities of the system.

In addition to the Action classes, the Controller also involves the use of Validators which serve to validate the input provided to the system. The Validators check the different input fields to ensure that they are of a valid format, and if not provide an exception which can be propagated and later displayed to the user so they know how to fix their input.

Model

The model encompasses all of the logic related to persistent storage in the system. Beans are utilized to store data related to different entities in the system, (e.g. a Clinician). Beans provide minimal functionality other than storing data.

The primary storage of the system is in a relational database. The database stores all persistent information, including the list of clinicians, preferences and other scheduling parameters which persistent between runs of the application.

In order to interact with the database, database access objects (DAOs) are employed. DAOs are java objects which interact with the relational database. They provide the ability to insert the data stored in the different Bean classes into the database. Additionally they provide methods to retrieve data from the database and store them in Beans which can then be utilized by the Controller to implement the business logic. Typically Action classes will use DAOs to store data and query data from the database. Each database entity is mapped to a single DAO and a common set of queries are provided for interacting with that entity within the DAO. The DAOs assume that the provided data is already validated and let the Action classes handle any exceptions. Connections to the DAO are accessed through the ConnectionFactory which has a singleton Connection instance which the application can utilize.

Overall, the design decision to go with the Model-View-Controller design pattern for our system helped to guide the design of the system. For most of the system each task had some associated database entity, which had an associated DAO and Bean to model that

entity. As a result the entities of the system were modeled in the database and DAOs and Beans were created to correspond with them. The Beans and DAOs were then utilized in the Action classes in order to provide the business logic of the application. The GUI was implemented using JFrames and JPanels consisting of various JComponents which invoked the different Controllers in order to provide the proper functionality. This allowed us to individually test the different components of the system. Both the Model and the Controller classes could be tested with JUnit tests and the View could be tested using a GUI testing framework. This design choice allowed us to more easily test the different components of our system and ensure that changes to the UI could be made without requiring major changes in the rest of the system functionality.

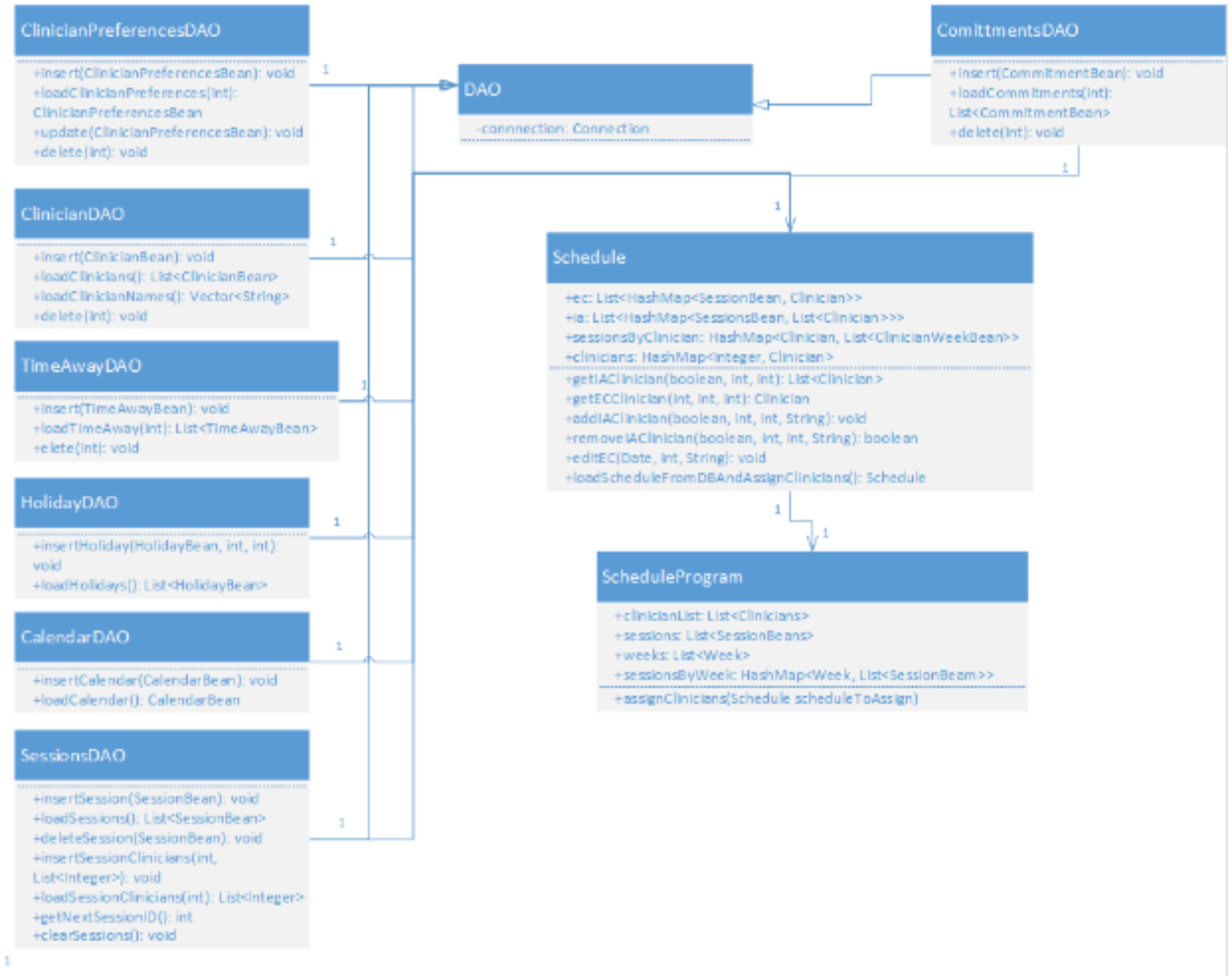

Fig1. Schedule Generation Class Diagram

The `Schedule` class acts as a mediator between the `DAO` classes and the graphical interface classes. Instead of having every class connect to the database via the `DAO` classes, all relevant information is stored within the `Schedule` class, and the `Schedule` class handles the communication to the database as well as the scheduling algorithm via the

`ScheduleProgram` class.

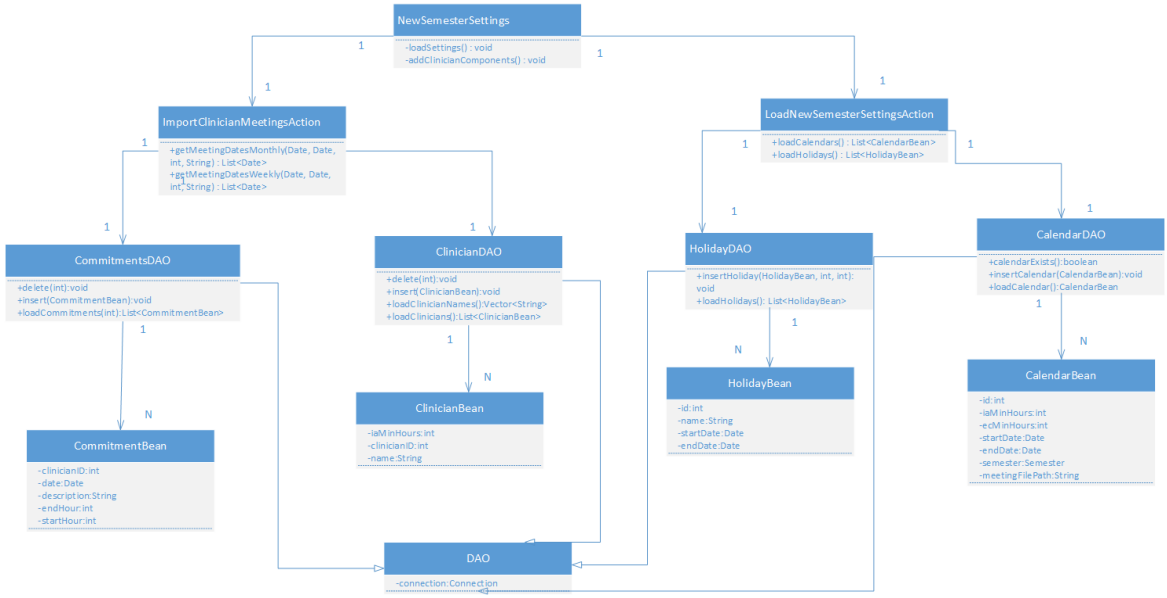

Fig2. New Semester Settings Class Diagram

New Semester Settings

At the beginning of every semester, an administrator will have to enter new settings for the system. The administrator can first enter in various settings to a form including dates for the semester, name of the term, holidays, and number of hours for IA and EC clinicians. They can also upload a file containing information for various meetings of a semester. The design of New Semester Settings is as follows:

View

The `NewSemesterSettings` class serves as the GUI for the user to enter the settings of a new semester and submit them. An upload button will upload a file using the `ImportClinicianMeetingAction` class. The submission is delegated to the `LoadNewSemesterSettingsAction` .

Controller

The Controller consists of the `LoadNewSemesterSettingsAction` and `ImportClinicianMeetingAction` classes. The `ImportClinicianMeetingAction` handles uploading a Microsoft Excel file of clinician meetings to the form. `LoadNewSemesterSettingsAction` handles the submission of the user input once its been validated to the persistent storage in the database. This involves interacting with the DAOs.

Model

The Model consists of the database schema, which is covered in the createTables.sql file. Specifically, the DAOs involved with a clinician's preferences are the `ClinicianDAO`, `CommitmentsDAO`,`CalendarDAO` and `HolidayDAO`. They interact with the database by

handling queries and inserting new or modified data. This data is modeled in the beans `ClinicianBean`, `CommitmentBean`, `CalendarBean` and `HolidayBean`.

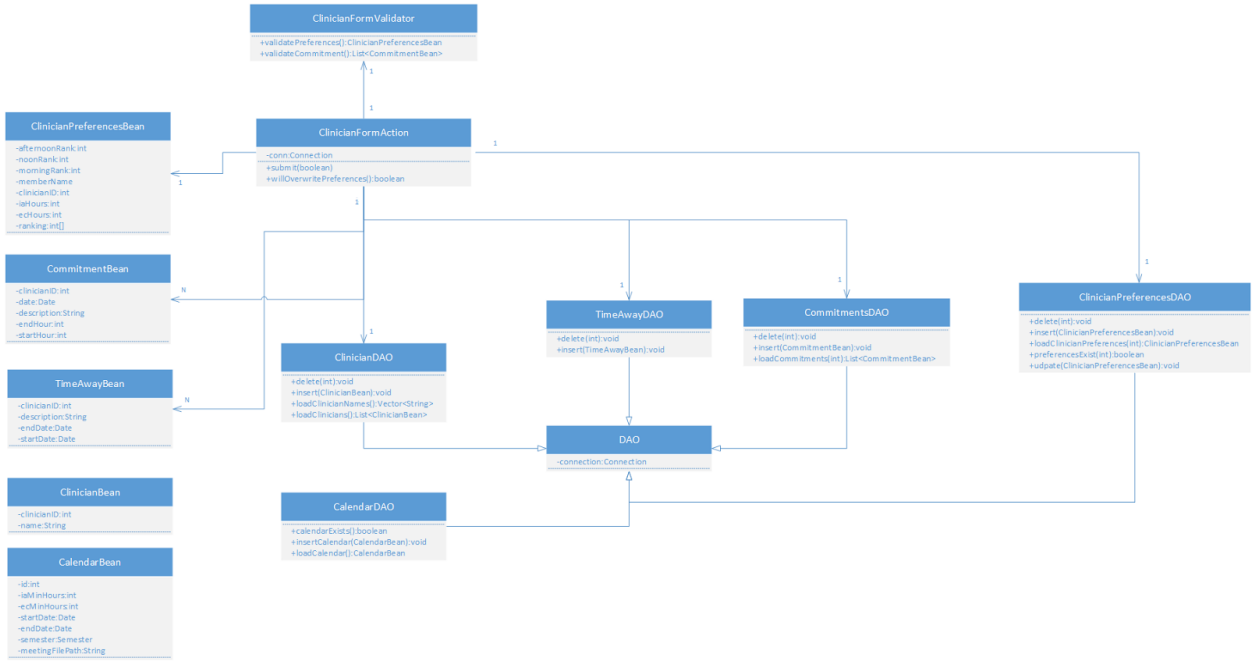

Fig3. Clinician Preferences and Clinician ID List Editor Class Diagram

Clinician Preferences

The Clinician Preferences form plays a large role in the system as all of the clinicians utilize it in order to submit their preferences to the system for scheduling purposes. The general flow of its usage is that a clinician can submit their preferences using the form, and they may also resubmit the form in order to update their preferences. From an admin's perspective they may load the preferences for a clinician to view them, and edit and resubmit them as they wish. The design of Clinician Preferences is as follows:

View

The ClinicianForm class serves as the GUI for the user to enter their preferences and submit them. When a form is submitted, the entered input is validated through the `ClinicianFormValidator` class. The submission is delegated to the `ClinicianFormAction`.

Controller

The Controller consists of the `ClinicianFormValidator` class and the `ClinicianFormAction` class. The `ClinicianFormValidator` handles validation of the clinician's preferences they input as well as their specified commitments for which they could not have any appointments scheduled. The `ClinicianFormAction` handles the submission of the user input once its been validated to the persistent storage in the database. This

involves interacting with the DAOs.

Model

The Model consists of the database schema which is covered in createTables.sql file. Additionally the various DAOs involved with a clinician's preferences are the `ClinicianPreferencesDAO`, `ClinicianDAO`, `CommitmentsDAO` and `ClinicianPreferencesDAO`. They are used to handle interaction with the database to query it as well as insert data. This data is modeled in the beans `ClinicianPreferencesBean`, `CommitmentBean`, `TimeAwayBean` and `ClinicianBean`.

Clinician ID List Editor

The Clinician ID List Editor provides the admin the ability to change the clinicians involved in a particular semester. They also have the ability to make changes to the clinician preferences. The design of the `ClinicianIDListEditor` is as follows:

View

The `ClinicianIDListEditor` class serves as the GUI for the admin to be able to add and remove clinicians. The admin may also edit the preferences for a particular clinician through this interface. Most of the validation is quite simple for add and remove functionality so it does not require a separate controller. However the editing clinician preferences is more involved and utilizes the `ClinicianLoadPreferencesAction` to handle the logic involved.

Controller

The Controller consists mainly of the `ClinicianLoadPreferencesAction` class. The `ClinicianLoadPreferencesAction` handles loading the preferences for a particular clinician and reuses the `ClinicianForm` for the admin to be able to edit their preferences.

Model

The Model consists of the database schema which is covered in createTables.sql file. The various DAOs involved with the Clinicians and their preferences are involved in the ID list Editor. These include `TimeAwayDAO`, `ClinicianDAO`, `ClinicianPreferencesDAO`, `CommitmentsDAO` and `CalendarDAO`. These handle interaction with the database to query it as well as insert data. This data is modeled in the beans `TimeAwayBean`, `ClinicianBean`, `ClinicianPreferencesBean`, `CommitmentBean`, and `CalendarBean`.

Future Plans

Our project is a specialized internal software customized for the Counseling Center's needs. While parts of it can be used for other purposes, it is mainly built to solve the Counseling Center's specific problems, so we would be unlikely to release it to the general public. Possible future plans include improving the interface and the quality of the scheduling algorithm, as well as adding features for the Counseling Center if they ask us. Besides that, we would try to make it more robust so the Counseling Center would not need to see us for fixes; most of us would not be available to maintain the code for the Counseling Center in the future.

Yusheng: I felt this project would have benefited from more upfront design and architecture. We had to make a few significant changes in later iterations because some parts of our programs were not integrating well. We also changed our algorithm to use an engine found online, which consumed more time than looking for similar engines or libraries early. In addition, we should have met with the Counseling Center earlier and more often as we had to make several changes later on after finding out that the Counseling Center's requirements would not what we originally interpreted. This project reinforced the idea that changes later in the software cycle cost more.

Nathan: The project provided all members a good opportunity to work with an actual client. We learned to reach out to a client early in the process in case they request a significant amount of changes to the project. We followed the extreme programming process well with regular meetings and always participating in pair programming. We also learned about the difficulty of coming up with scheduling algorithms from scratch and to utilized third party software whenever possible to save time. In the end we accomplished all of our goals for the project and are satisfied with the results.

Jeffrey: The process of Extreme Programming over the course of the project demonstrated the notion that each development process has its own strengths and weaknesses. Dividing up the user stories among pairs allowed the work to be split up in a reasonable way, but it often lead to conflicts in designs between different pairs that worked on components that need to be integrated together. I think that in this regard it would have been more effective for us to have invested some more time in the general system architecture from the beginning. A fair amount of time was required to integrate the different components of our system at the end. The usage of third party libraries played an important role in completing our user stories in a reasonable amount of time because it prevented us from having to write our own excel file parser among other things. One of the more useful practices was the thorough usage of both GUI and unit tests over the course of the project which gave me a lot more confidence to go through and refactor sections of code. This allowed our design to evolve with the project and us to improve the overall quality of the code because it was easy to detect if something was broken, and then we could inspect what tests started to fail and react accordingly.

Denise: This project clearly demonstrated the value of good communication in software engineering, both with clients and with teammates. It can be difficult to convey questions about engineering requirements (e.g. available tools, what software applications the clients are familiar with, etc.) with non-technical clients, and it can likewise be difficult for the clients to convey their constraints and requirements to an engineer. On the teamwork side, it is

oftentimes challenging for every team member to stay aware of the progress, goals, and time constraints of the rest of the group. Our team encountered examples of all of these obstacles throughout our work. The XP process provided solutions to all of these problems, and having this experience will definitely help each of us preempt these issues in the future.

Kevin: I learned a lot from working on this project with my fellow classmates. Most notably, how important communication and planning was in the development process. Our group communication was great, but we lacked a bit in the planned department. Maybe it was due to the nature of extreme programming or due to our poor planning, but we had to change significant portions of our design throughout the course of the semester. And dealing with those changes was difficult because we were mostly working in pairs. When one pair wanted to change something about the overall architecture, they had to wait until we had a team meeting before we could make any changes because of how everything was connected through the database. I learned about the importance of making sure to comment and refactor often because not everyone codes the same way, and making the code easier to understand helps your teammates.

Ryan: Developing the scheduling application for the Counseling Center was very appropriate for this class. We were able to meet with the client in person, and I think that the real-world experience we got in adjusting to her descriptions of the requirements was very valuable. That process was also a reminder that communication with the client is key, especially when the user needs to convey particular technical requirements (for scheduling) despite not having a computing background. As for the XP process, I think that the biggest concern was project management; without a manager or a detailed design document during the process, we sometimes had struggles both with designing the high-level infrastructure of the system and with communication between pairs while implementing our code. These issues were mitigated by unit testing and regular meetings, and were mostly sorted out before the final iteration's refactoring; however, the importance of regular communication between team members could not be understated.