CSCI 4448: Object-Oriented Analysis and Design: Project Part 6

* Note: auto-generated documentation for the project is located in the Walton_ClubManagementSystem_Part6_Code directory. Open the index.html file and click on "Module Index" to see documentation for all modules.

1. **Name:** Lindsay Walton

2. **Project Description:** Club Management System
   Club Management System is a club management program that allows club officers to keep track of members, officers, and events. Users can create new events, create tasks for events, and assign tasks to officers/members so that they can track the progress of the event. Officers can also log member attendance for the event. Club Management System is written in Python and uses PyQt and a model-view-controller design.

3. **Features that were implemented:** General user requirements:

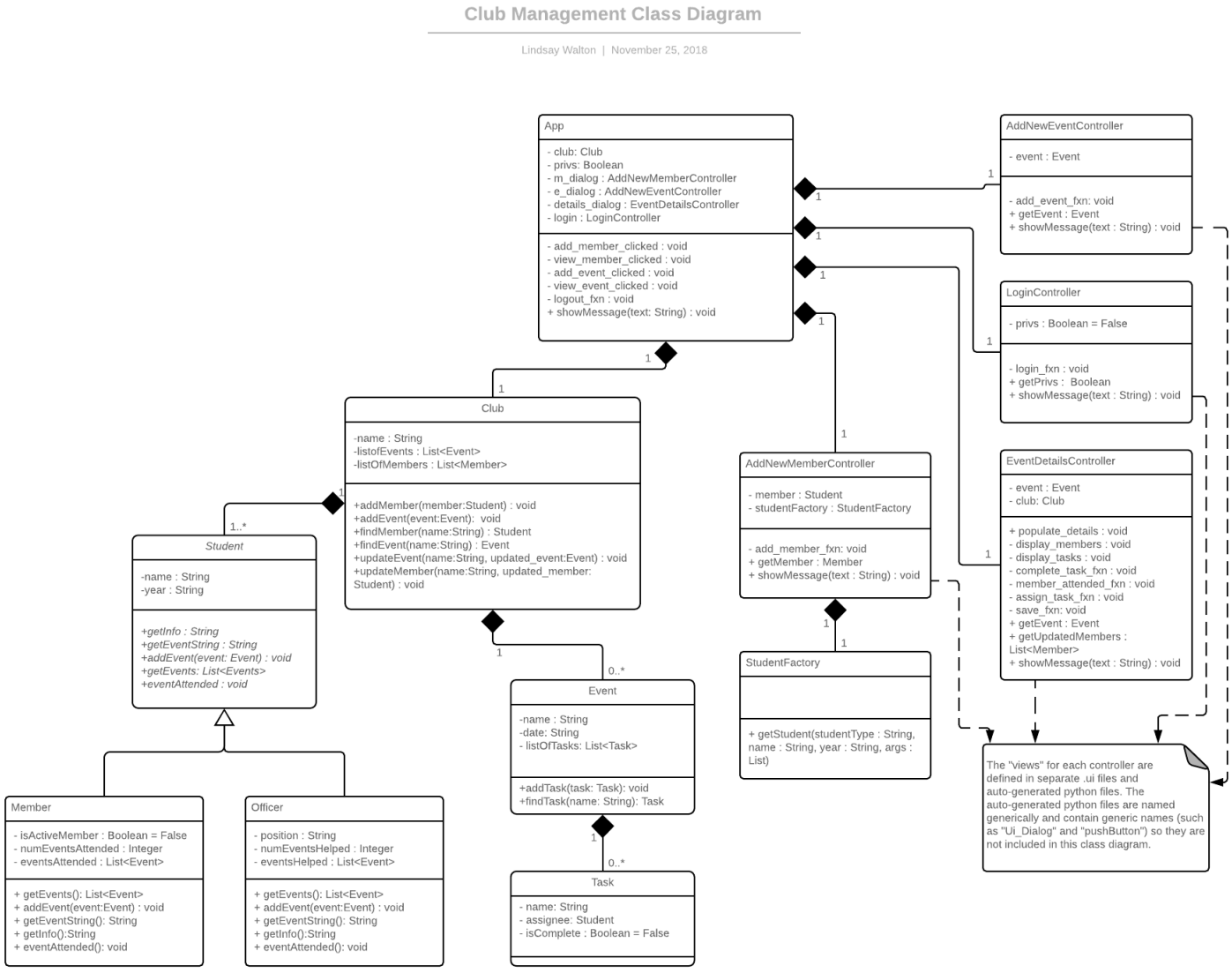
ID	Actor	Requirement
U1.0	User	User shall be able to log in.
U2.0	User	User shall be able to view members.
U3.0	User	User shall be able to view events.
U4.0	User	User shall be able to view tasks for events.
U4.1	User	User shall be able to view status of events (preparation complete).
U5.0	User	User shall be able to view active member status.
U5.1	User	User shall be able to view how many events he/she has attended.

   Officer requirements:

ID	Actor	Requirement
O1.0	Officer	Officer shall be able to add a new member.
O1.1	Officer	Officer shall be able to add a new officer.
O2.0	Officer	Officer shall be able to create a new event.
O3.0	Officer	Officer shall be able to create a new task for an event.
O3.1	Officer	Officer shall be able to assign a task to a member.
O4.0	Officer	Officer shall be able to indicate that a member attended an event.
O5.0	Officer	Officer shall be able to view active member status of all members.

4. **Features that were not implemented:** None. All features were implemented.

5. **Class Diagram**



Note: to reduce the size of the class diagram to allow it to fit on this page (and to be legible), all constructors, getters, and setters have been removed. Assume that each class has a constructor and each class variable has a getter and a setter.

What Changed:

(a) Splitting the original singular controller into multiple controllers for each use case: adding a new event, viewing/editing event details, adding a new member, and login. The single controller was getting far too complex and hard to follow, and it violated the single responsibility principle, so I created separate controllers.

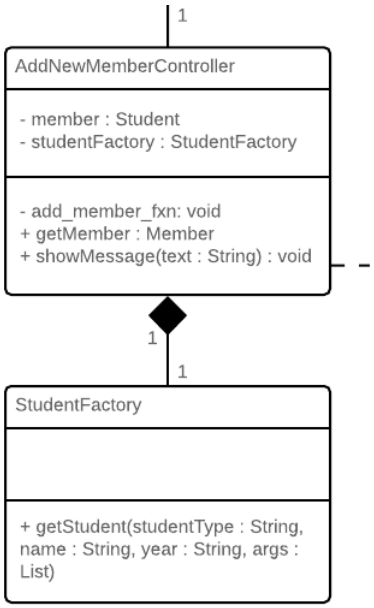
(b) Splitting the original singular view into multiple views for each use case. Similarly to the controllers, a single view was getting far too complicated and hard to manage. I created 5 separate views instead to manage each use case.

(c) Creating a top-level App class. The App class acts as the interface between the models, views, and controllers. I had not originally planned for this class, but due to the structure of PyQt, it was a necessary class.

(d) Various functions were added throughout development. For example, the "getEventString()" method was added for the Student, Member, and Officer classes (an abstract method in Student). When

marking attendance for an event, I wanted to display "attended" for regular members, and "helped with" for officers (because officers help run events). Rather than clogging the view and controller code with if-statements to determine which string was necessary, I created the abstract method in Student that each sub-class could implement to return their necessary string. All of the abstract methods in Student have similar purposes.

6. **Design Pattern:** Factory



The AddNewMemberController class uses the Factory design pattern. The design pattern itself is implemented in the StudentFactory class. The StudentFactory class returns a type of Student, either Member or Officer, based on the type of the Student, which is a parameter for the StudentFactory method.

In the AddNewMemberController, a user can create a new member: either an officer or a regular club member. In the initial implementation of the code, there was an if statement similar to the following pseudocode:

```
if new_member is officer:
    member_object = Officer(args)
if new_member is member:
    member_object = Member(args)
```

For only two types of students, this code is not awful. However, it violates the "open for extension, closed for modification" principle of object-oriented design. This code is not easily extended. If new Student types are needed, such "board member" or "advisor", the controller code would get clogged with if statements determining the type of student that needs to be created, which could easily become long and complex. In addition, it violates the Single Responsibility Principle. The AddNewMemberController code is no longer just performing its intended purpose of handling user interaction for creating a member. It is also overlapping with the "model" of the Model-View-Controller and determining which type of Student needs to be created. The controller code needs to serve the purpose of handling user interaction, and nothing else.

To solve these problems, I decided to use the Factory design pattern and create a StudentFactory class. The StudentFactory has a "getStudent" method which takes an argument of which type of student needs to be created and handles the creation in the class. The controller simply has to create an instance of the student factory, and get the student by passing the student type to the StudentFactory. This makes the code easily extendable by creating additional student types and adding them to the StudentFactory

class. In addition, the controller code now handles only its single responsibility of user interaction. The Factory design pattern can also be implemented elsewhere in the code if a user wants to be able to have different types of events or different types of tasks.

7. *What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?*

   I have learned the importance of fully designing a system before attempting to implement it. It may seem easier to create a general overall plan and figure out the details as you go along, but I quickly discovered that that approach was misguided. For my initial design, I planned to use a single controller and a single view, and decided that if issues occurred with this approach, I could add additional views and/or controllers later. This proved to be a mistake. By using only a single view and a single controller, everything was highly coupled. The code was complex and hard to follow. It also violated the Single Responsibility Principle. By having a single controller that handled all user interactions, the code was simply too complicated to follow, develop, and debug.

   When I realized that this approach wasn't going to work, it was difficult to decouple the objects and separate the project cleanly into different controllers and views. I finally decided to simply start over: create new views and new controllers from scratch. If I had designed the project fully before attempting to implement it, I would have realized this mistake much sooner and not wasted time and effort by trying to implement a faulty system. I also learned the importance of the Single Responsibility Principle. When a module or class has multiple responsibilities, the code quickly gets too complicated to follow and project development gets delayed.

   Along the same lines, I have also learned the importance code analysis and refactoring. It is essential to be able to accurately analyze your own code and realize the flaws in your design. A developer needs to be able to notice and accept when his or her design is faulty and take steps to improve it. I realized that my single view and single controller design wasn't going to work: the code was getting too cluttered and complicated. Even though I had put a lot of time into this design, I recognized that the current code needed to be thrown out and I needed to start over; otherwise, the system would not have functioned correctly and debugging, testing, and documentation would have been unnecessarily complicated. Analysis and code refactoring are essential steps in the design process.