Software Design Patterns CA Documentation

Law Management Software

Diogo Moreira

T00191631

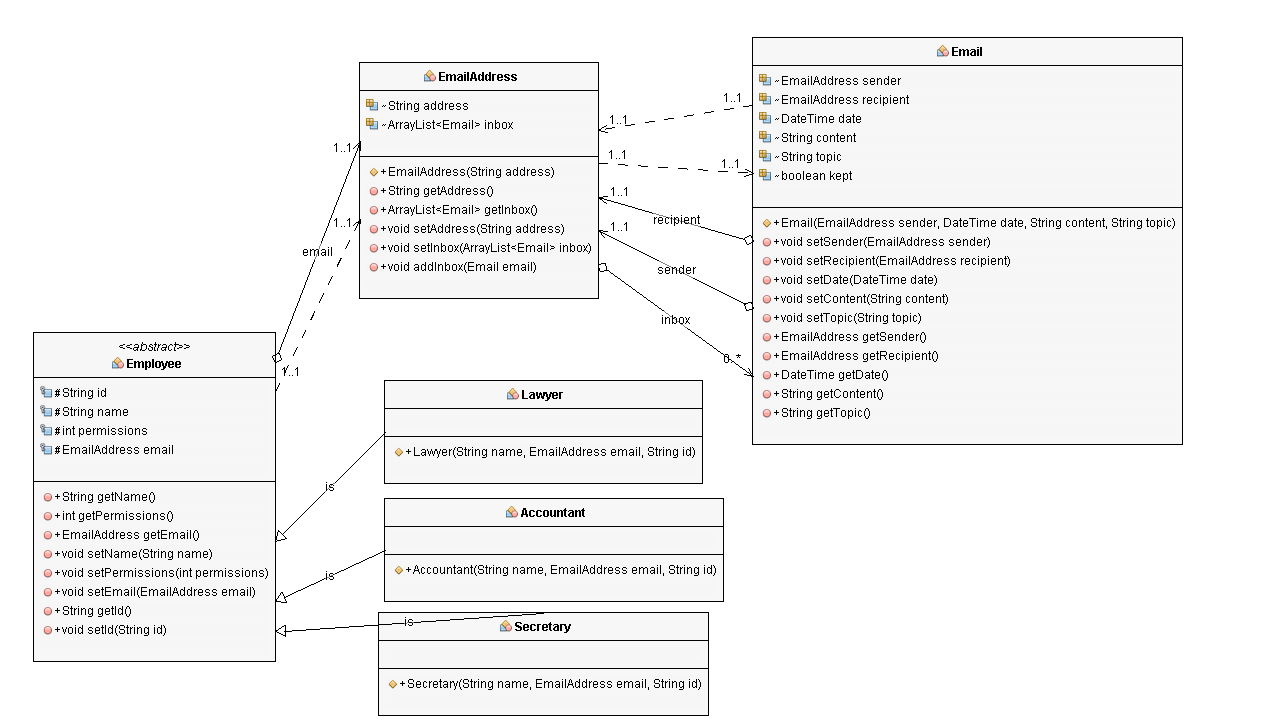
Table of Contents

1. Introduction----------------------------------------------------------------Page 3
2. Structure of Code---------------------------------------------------------Page 3
3. Patterns not Implemented --------------------------------------------Page 8
4. Pattern Implemented---------------------------------------------------Page 9
5. Using Program------------------------------------------------------------Page 12
6. *Introduction*

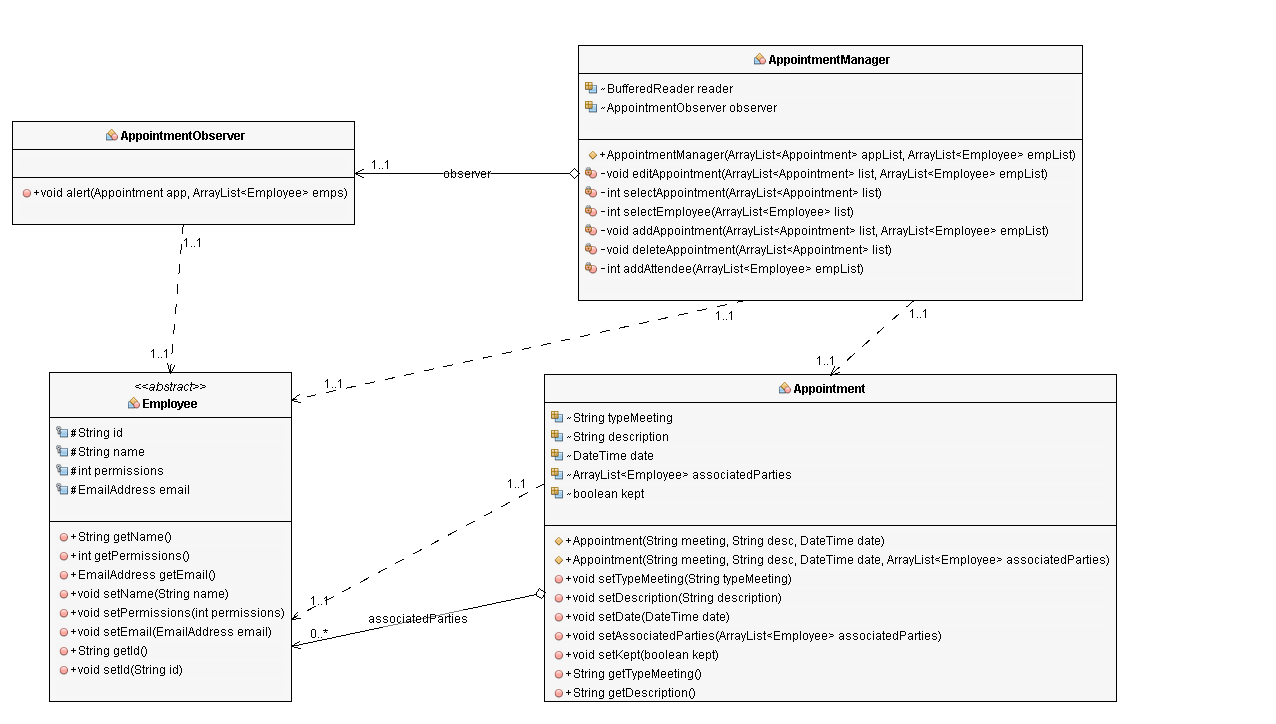
Law-Management-Software is used in many starter, and larger law firms to help organize everyone. From organizing someone schedule, to keep some aware on what they have to do, to sending emails. This software has many other utilities, most have the ability to do bookkeeping, and document creators, to help print out those documents so they don’t have to be typed up every time. It fits the needs of legal practices and tackles the challenges that every legal professional must deal with.

This sort of software, having many functions, was a perfect fit to dissect, and employ my understanding of software design patterns. This paper will discuss my implementation, and the design patterns I tried to implement, and the ones I did implement. I will discuss why these designs were either valid, or not, and compare code that was before the implementation of these patterns, and after implemented. I will also discuss the structure of my code.

1. *Structure of Code*

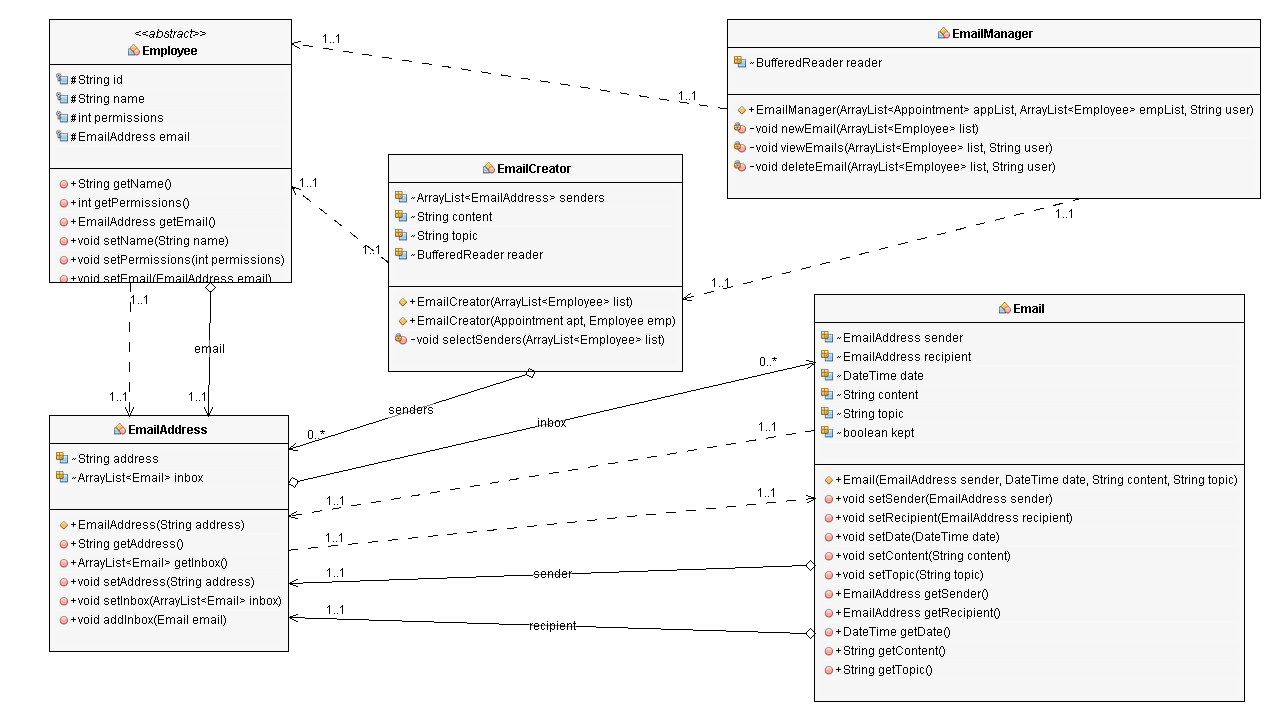


This UML diagram shows the structure of how we hold information about our employees. Our superclass, Employee holds the different information about the employee, key fields are email address, id, and permission. EmailAddress holds all the emails for that employee, makes it easy to organize who has what email. The field id’s purpose is for login information. When logging in, the id is compared to the id of all employees, and when there is a match, determines what employee object the user is. Although permissions were never implemented, its main purpose was to if necessary restrict an Employee object, on what was available to them. It would also be a feature for the email system, where someone could send an email, to the objects with the right permission. The permissions were the main reason I created the subclasses of Employee.



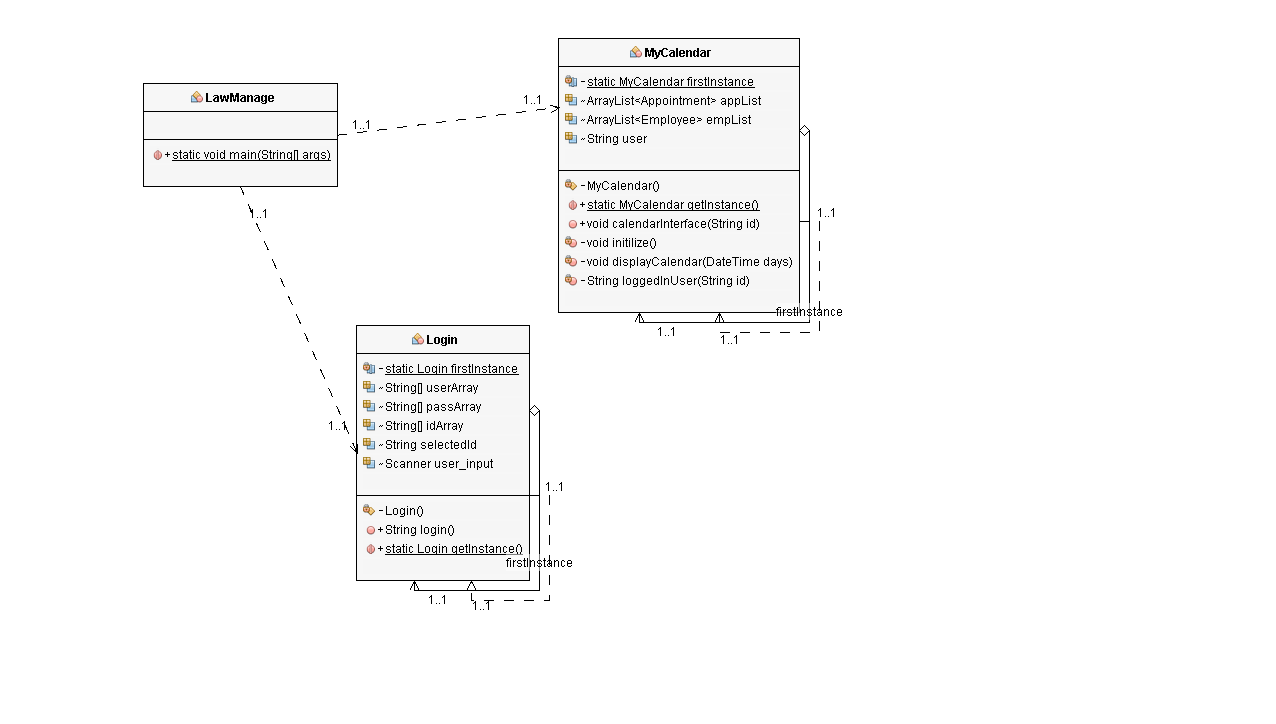
This is an UML diagram of the Appointment management system, and how we work with appointments. Appointment holds relevant information like the kind of appointment, the details of it, the date, and an arraylist of the associated employees related to the appointment. The arraylist of employees is to have easy access to the employee objects that are related to the appointment object, so when something changes to the appointment, an email can be sent to all parties associated with the appointment.

Appointment manager class acts as an interface to appointment objects. If there needs to be changes to appointment objects dynamically, the AppointmentManager class handles it. The class creates, deletes, and edits appointment objects. Once the AppointmentManager class has performed an action to an appointment object, it alerts the AppointmentObserver class of it, and the AppointmentObserver notifies all the associated parties about the change, by sending an email to all the employee object’s EmailAddress field.



This is an UML diagram of the Email management system, and how we work with emails. Like explained earlier, EmailAddress array lists hold emails for each individual employee object. The EmailManager class, similar to the AppointmentManager class, handles the creation, deletion, and viewing of emails. When Email Manager want to create an email object, it calls the EmailCreator class to handle it. Following the Single Responsibility principle, the Email Creator class is separate from the EmailManager class because a class should have a single purpose. I also did it because I did not want AppointmentManager class to talk to EmailManager class for creating email objects.

LawManage handles the Login class, and the Calendar class. If Login is successful, it returns an id, which is used to determine what employee the user has signed in as. The MyCalendar class acts as the calendar, and the menu for using other functions.



1. *Patterns Not Implemented*

Factory pattern

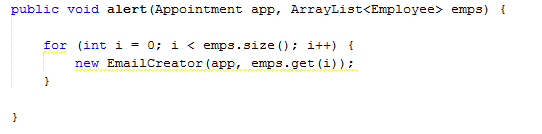
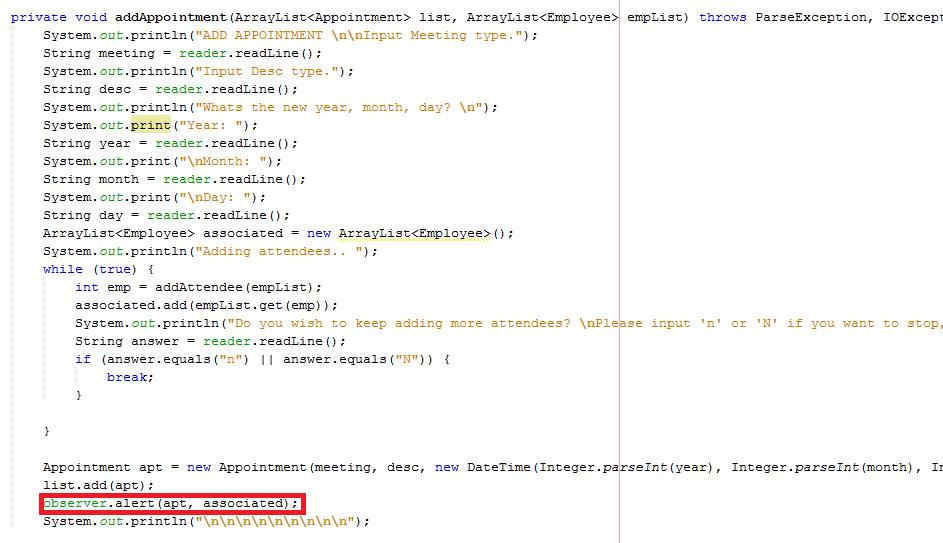
Although it was considered, the factory pattern ended up not being used in the final product of the program. The factory pattern defines an interface for creating an object, it also allows you to encapsulate object creation so that you can keep all object creation code in one place. I considered implementing this pattern for the initialization of my employees. But decided it would not have added anything to the program. In the overall scope, it would not have contributed to the quality of the program.

Strategy Pattern

Another pattern I did not include in my final product is the strategy pattern. The strategy pattern defines a family of algorithms, encapsulate them, and makes them interchangeable. It allows the behaviors of a class to be changed dynamically. This pattern did not make it in the final product because I did not believe it would fit properly into my program. At this point in the program, there is no reason to integrate it, as the behaviors of the Employee class did not need to be changed dynamically.

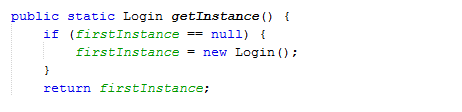
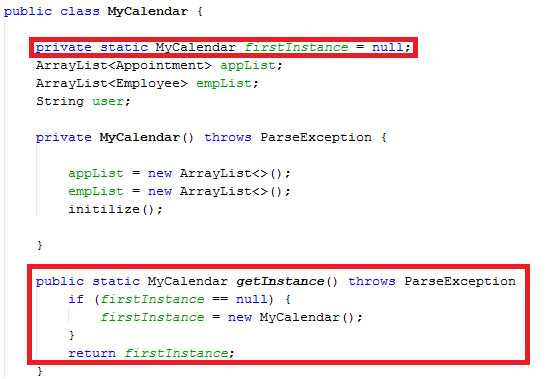
1. *Patterns Implemented*

The first pattern I implemented was the Observer pattern. This pattern’s is used when an object, maintains a list of its dependents, called observers, and notifies them automatically of any state changes. A good example of systems where the observer pattern would be a good fit would an distributed event handling system. Many GUI frameworks work well with the observer pattern.



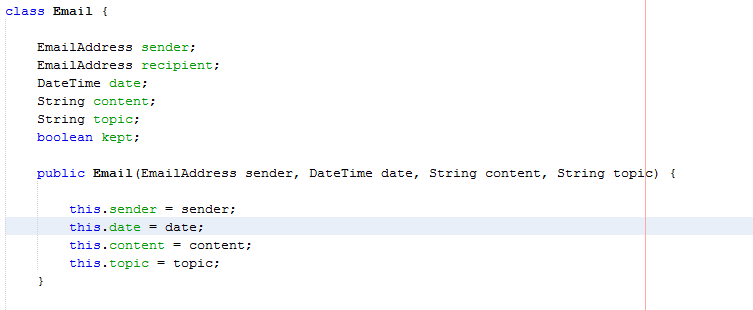
Using my addAppointment method as an example, when I add an Appointment, any Employee objects that are associated with this appointment, will be sent to the observer method. The observer method will then call the Email Creator class, which will create emails for all the employees.

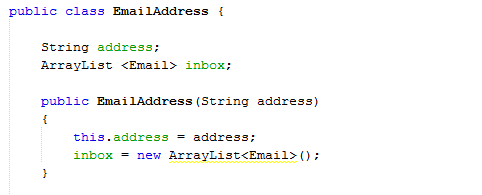
The Singleton pattern is another pattern I have incorporated into my design. The Singleton pattern ensures that there only ever one instance of a class, and that it can be accessed globally. This pattern is usually used for loggers, or service locators.



Since the MyCalendar class initializes employees and appointments, it cannot be recreated every time a user needs to get off, and switch accounts, so to prevent that, I used the singleton pattern. This way, if someone logs off, when the new user logs on, all the information will still be there. The Login class uses this information to keep track to of who has logged in, if the instance was being recreated every time, it wouldn’t be able to keep track.

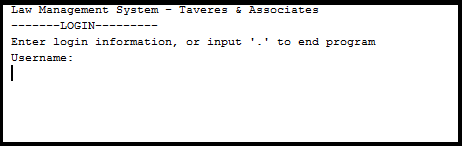
The Composite pattern builds structures of object in the form hierarchy trees, that contain both compositions of, and individual objects. This pattern can be used for many applications that have to deal with hierarchical data.





In my program, I had reach employee have a field of EmailAddress, which itself contained an arraylist of Email objects. This is a classic use of the composite pattern, where we have an hierarchy structure of EmailAddress objects, with many individual Email objects.

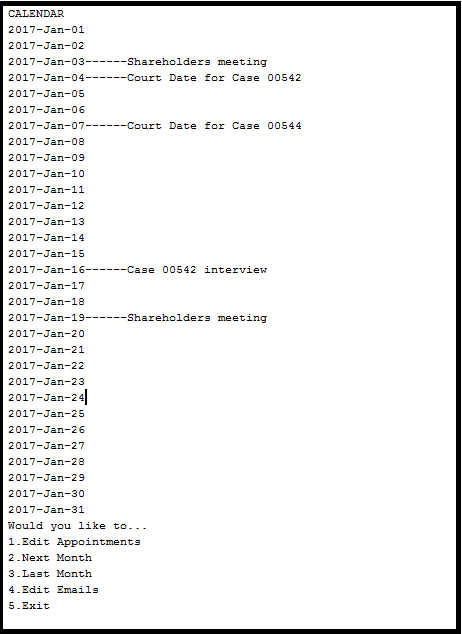
1. *How to use Program*
2. Start the LawManage class, and you should be introduced to this. If you wish to end the program, input ‘.’.



Here all the login credentials -

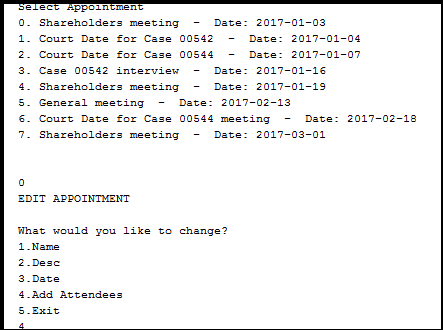
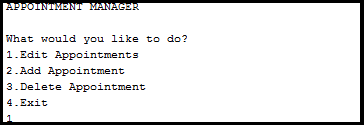
|  |  |
| --- | --- |
| Username | Password |
| lawyer1 | lawyer |
| secretary1 | secretary |
| accountant1 | accountant |

1. After successfully logging in, you will be introduced to the calendar.



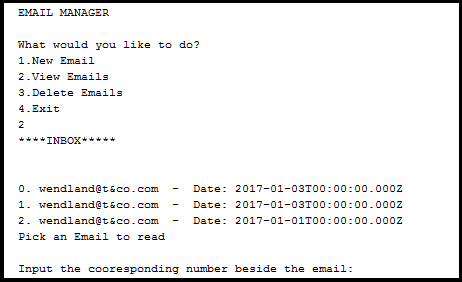
The calendar shows the all the dates of the month, as well as any appointments on those dates. Inputting 2, or 3 will allow you to navigate the different months of the calendar. If you wish to return to the login screen, input ‘5’.

1. If you Input 1, it will bring you to the appointment manager.



Appointment manager gives you many options to interact with appointments. You can add, edit, or delete appointments. Just input the number beside the option you want tot use, and follow the steps. If you click edit appointment, you will be asked which appointment you want to edit, again, input the number beside the appointment you want to edit. It will bring you another menu, to let you choose which aspect you wish to change. Input 5 to stop editing.

1. From the main menu, if you wish to access emails, just input ‘4’ to access it.



You will be given 3 options, creating, viewing, or deleting emails. If you wish to view, or delete emails, it will bring you with all the emails for that user. Just input the number that corresponds with that email. Else just follow the instructions given to complete the task.