**Take Home Java Coding Project**

Total time spent:8.5 hours

**Task 1:**

The solution I designed contains 3 main components whose interfaces are defined in their the corresponding .java file names:

* **CalendarManager:**
  + Responsible for holding days that have been selected as holidays/vacations or having events scheduled for them. Also processes requests to count the occurrences of a proposed weekly meeting time. This is the main component that holds two different implementations of the meeting scheduler.
* **Day:**
  + Days are meant to represent days off or days that have an event scheduled. After creation, they are added to a list in the CalendarManager. This is the main way of representing holidays, vacations, etc.
* **Event**:
  + Events have two sole purposes:

1. In the event scheduler implementation, they are inserted into an EventTree, which holds all the events for a certain day.
2. When passing in a request to count the occurrences of a meeting, the meeting itself is an Event type that is simply used to check for collisions with other meetings on a certain day or to see if that Event’s date collides with a day off in the CalendarManager.

*Developer Interaction*:

Developers would use my design (depending on implementation preferences) as a means of processing user requests for meetings. For example, a GUI developer would create a feature for allowing a user to create a weekly meeting. The submission of the user’s choices would be sent to the CalendarManager, which would count the number of occurrences based on other events that are scheduled, holidays, or vacations. The CalendarManager would finally report back the number of occurrences.

Developers could also use the event scheduling system as a basis for scheduling in the calendar’s GUI. A good example of a use case would be using the EventTree to visually display a user’s schedule for a specific day.

**Task 2:**

First and Foremost: Much of the implementation for this design relies on

Java’s ***Calendar*** and ***Date*** classes as a means of doing calculations with dates.

***Event Scheduling implementation*:**

This implementation involves each Day having its own BST style EventTree, which represents a schedule of events for that day. When processing requests for a reoccurring meeting, the meeting is passed as an event and traverses the EventTree looking for a collision between intervals of time. This implementation would be useful if a user’s preferences were to take their daily schedule (as well as other people’s schedules) into account when attempting to schedule a weekly meeting. This could also easily represent holidays/vacation by having the time interval span the entire day.

***Days Off implementation:***

For this implementation, I simply add a new Day to the CalendarManager for every day someone is on vacation, holidays, etc. When counting the occurrences for a weekly meeting, a boolean value within the Day object that indicates a Day being off or on is used to determine if the requested meeting will collide with this Day.

Complexities:

Since a BST tree’s search is log(n) and we must visit each day between a start date and an end date (O(n)), the time complexity of the Event Scheduling implementation of counting occurrences is O(nlogn). Since we are not dealing with the binary search tree for the Days Off implementation, the time complexity is O(n). Although the Days Off implementation is faster, it does not provide some of the useful features that the Event Scheduling implementation does.

**Task 3:**

A demonstration is documented in the FrontEnd.java file.