Sahaj software | inspiring brilliance

Guidelines

At Sahaj, we strive to build high-quality software that has strong aesthetics (readable and maintainable), has extensive safety nets to safeguard quality, handles errors gracefully and works as expected, without breaking down, with varying input.

We are looking for people who can write code that has flexibility built in, by adhering to the principles of Object Oriented Development, and have the ability to deal with the real-life constraints/trade-offs while designing a system.

It is important to note that we are not looking for a GUI and we are not assessing you on the capabilities around code required to do the I/O. ***The focus is on the overall design****. So, while building a solution, it would be nicer if the input to the code is provided either via unit tests or a file. Using the command line (for input) can be tedious and difficult to test, so it is best avoided.*

Following is a list of things to keep in mind, before you submit your code, to ensure that your code focuses on attributes, we are looking for -

- Is behaviour of an object distinguished from its state and is the state encapsulated?
- Have you applied SOLID principles to your code?
- Have you applied principles of YAGNI and KISS (additional info here)?
- Have you unit tested your code or did TDD? If you have not, we recommend you read about it and attempt it with your solution. While we do not penalise for lack of tests, it is a definite plus if you write them.
- Have you looked at basic refactoring to improve the design of your code? Here are some guidelines for the same.
- Finally, and foremost, are the principles applied in a pragmatic way. Simplicity is the strongest of the trait of a piece of code.

Sahaj software | inspiring brilliance

Problem Statement

(Average time to write a solution 5-8 hrs)

Design and implement a library to represent a schedule of recurring events. The pseudo-code below shows typical usage of the library:

```
recur:Schedule = Schedule.for(<Schedule specification>)

print("Schedule: "+ recur.getAllOccurrences())
```

Features, to be supported, by the library, are summarized in the following APIs -

```
interface Schedule {
	startDate():date;
	endDate():date;
	getOccurrences(limitNumberOfOccurences: int): list of dates
	getOccurrencesFrom(startDate:Date, numberOfOccurences:int):list of dates;
	getAllOccurrences():list of dates;
	getNumberOfOccurences():int // return #of occurrences for bounded schedules.
}
```

The behaviour of the library is similar to scheduling a recurring event on a calendar with start date, end date, and a repeat schedule.

Following are some of the examples of different kinds of recurring schedule use cases that need to be supported:

1. Every second Saturday is a holiday.
2. Remind me to pay my phone bill on the 10th of every month.
3. 2nd Sep is my anniversary.
4. Every Tuesday and Thursday is team catch-up.
5. Every 1st and 3rd Sunday, I need to visit the hospital.
6. 2nd Dec 2017 we have a school reunion. (non-recurrent event)
7. Every alternate Wednesday our sprint ends.
8. Once in 2 months, on the 10th I need to pay my credit card bill.
9. Once in every quarter, 5th we have shareholders' meeting.

Sahaj software | inspiring brilliance

Tasks

1. The first step is to design a schedule format for providing schedules (listed above) and write the DSL or parser for the same. Schedule format need not be free-form text. It can be any format you are comfortable with, e.g. comma separated, JSON, YAML, your language of choice, etc. An example of using a map of key-value pairs with JSON format is as follows -

    - For Schedule: *Stand-up Everyday*, input should be given as
      `{eventName:"Stand up", repeat:"daily"}`
    - For Schedule: *Anniversary every year*, input should be given as
      `{eventName:"Anniversary", date:"2nd", month:"Sep", repeat:"yearly"}`
    - For Schedule: *Weekly Catch-up*, input should be given as
      `{eventName:"Catch-up", day:"Tuesday, Thursday", repeat:"weekly"}`
    - For Schedule: *Monthly Second Saturday Holiday*, input should be given as
      `{eventName:"Holiday", ordinal:"2", day:"Saturday", repeat:"monthly"}`

2. In the next step, implement the API that takes the schedule specification, designed in the previous step, and produces a **Schedule** object

    ```
    schedule:Schedule = Schedule.for('{eventName:"Stand up", repeat:"daily"}')
    ```

3. Now implement rest of the API of the `Schedule` interface.

    ```
    print("Stand-ups are on: "+ schedule.getAllOccurrences())

    print("First 3 Stand-ups are on: "+ schedule.getOccurrences(3))

    print("Stand-ups start from: "+ schedule.startDate())

    print("Stand-ups end on: "+ schedule.endDate())

    print("Number of meetings: "+ schedule.getNumberOfOccurences())
    ```

4. Implement unit tests to showcase the various kinds of schedules (9 schedules are shown in the previous section), supported by your library. The program must not accept any console input.

Submission

1. All build infra scripts - to compile and run the submission.
2. Any additional documentation or instructions for the reviewer.