SESSION 2023/2024 SEM 2
SECJ1023 ProgrammingTechnique II

PROJECT 1:
SLEEP CYCLE ANALYZER SYSTEM

TASK:
GROUP PROJECT DELIVERABLE 2
PROBLEM ANALYSIS AND DESIGN

LECTURER:

Dr. Lizawati binti Mi Yusuf

<u>Group members:</u>

NUR SYAKIRAH ADILAH BINTI AZRI A23CS0159
YEO WERN MIN A23CS0285
VIBHUSHA A/P SAMPASIVA RAO A23CS0194
JELIZA JUSTINE A/P SEBASTIN A21EC0034

Section A: Flow Chart

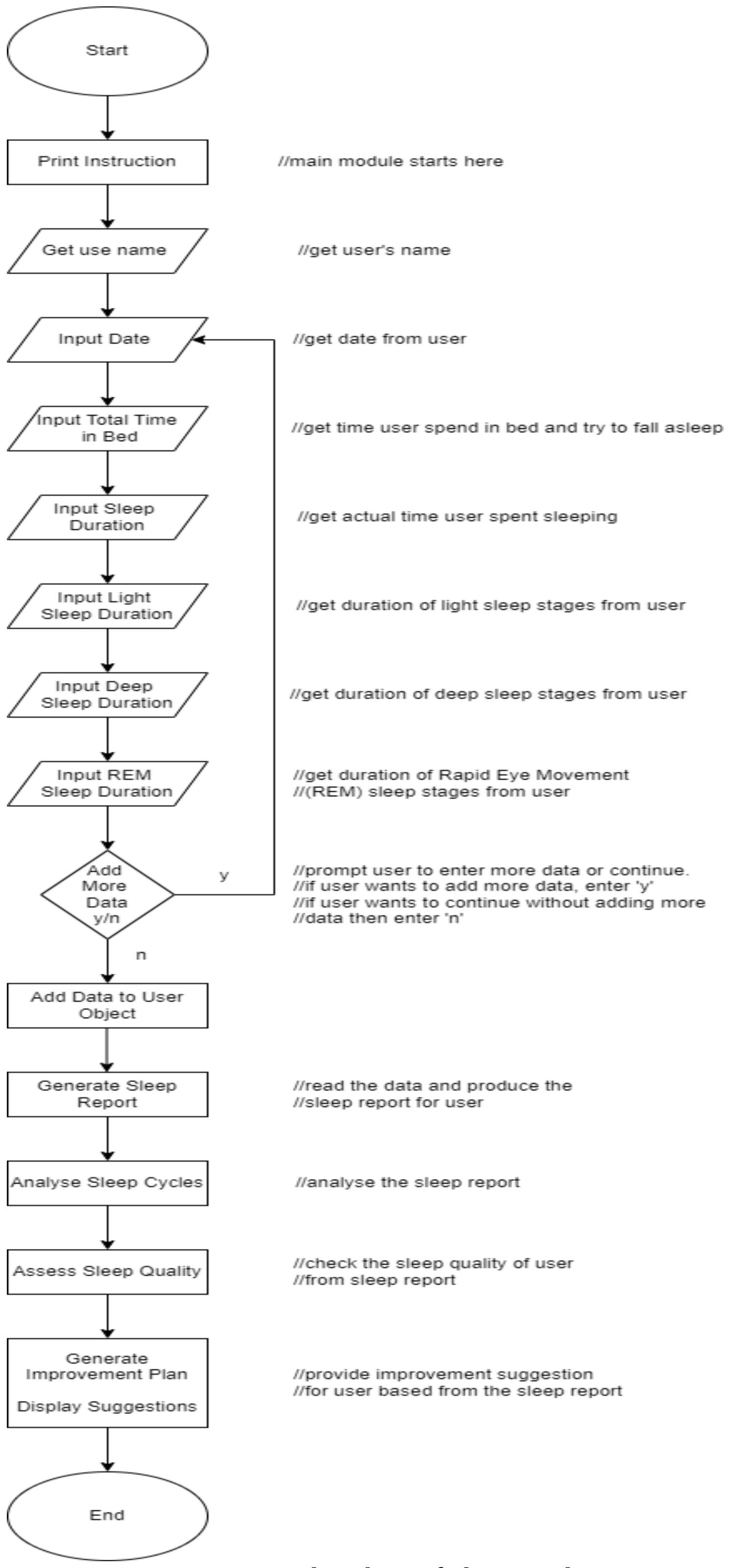
Start
Print Instruction
//main module starts here
Get use name
//get user's name
Input Date
//get date from user
Input Total Time in Bed
//get time user spend in bed and try to fall asleep
Input Sleep Duration
//get actual time user spent sleeping
Input Light Sleep Duration
//get duration of light sleep stages from user
Input Deep Sleep Duration
//get duration of deep sleep stages from user
Input REM Sleep Duration
//get duration of Rapid Eye Movement
//(REM) sleep stages from user
Add More Data y/n
y
//prompt user to enter more data or continue.
//if user wants to add more data, enter 'y'
//if user wants to continue without adding more
//data then enter 'n'
n
Add Data to User Object
Generate Sleep Report
//read the data and produce the
//sleep report for user
Analyse Sleep Cycles
//analyse the sleep report
Assess Sleep Quality
//check the sleep quality of user
//from sleep report
Generate Improvement Plan
Display Suggestions
//provide improvement suggestion
//for user based from the sleep report
End

Figure 1: Flowchart of Sleep Analyzer System

The flowchart represents a sleep analyzer system that assists users in tracking and analyzing their sleep patterns. The program begins by printing instructions and prompting the user to enter their name. It then asks the user to input various sleep data, including the date, total time in bed, sleep duration, light sleep duration, deep sleep duration, and REM sleep duration. It then enters a loop where the user can decide to add more data collected from different data or stop entering more data. After collecting each set of data, the program adds it to the user's sleep records.

Once the data collection phase is complete, the program generates a sleep report summarizing the data entered. It then proceeds to analyze the sleep cycles and assess sleep quality based on the recorded information. Finally, the program generates an improvement plan, offering suggestions to enhance the user's sleep quality. The program concludes by displaying these suggestions and terminating the execution.

Section B: Problem Analysis

(i) Classes involved, Attributes, Methods

1. User

   - Attributes: userID (*string*), name (*string*), age (*int*), weight (*double*), height (*double*), sleepSessions (*vector<SleepSession>*), sleepReport (*pointer to SleepReport*), sleepQualityAssessment (*pointer to SleepQualityAssessment)*, suggestions (*vector<pointer to Suggestion>*) ,sleepCycleAnalyzer (*SleepCycleAnalyzer*)

   - Methods: updateProfile(*name: string*, *age: int*, *weight: double*, *height: double*), analyzeSleep(), qualityAssessment (), generatelmprovementPlan()

2. SleepData

   - Attributes: timestamp (*string*), duration (*double*), deepSleepDuration (*double*), lightSleepDuration (*double*), REMduration (*double*), session (*SleepSession*)

   - Methods: getTimestamp(), getDuration(), getSession()

3. SleepSession

   - Attributes: sessionID (*string*), startTime (*string*), endTime (*string*), sleepData (*vector<pointer to SleepData>*)

   - Methods: getSessionDuration()

4. SleepQualityAssessment

   - Attributes: sessionID (*string*), qualityScore (*double*), description (*string*), sleepData (*vector<SleepData>*)
   -
   - Methods: assessQuality(), getQualityScore(), getDescription()

5. SleepCycleAnalyzer

   - Methods: analyzeSleep(*user:User*), qualityAssessment(*user:User*)

6. SleepReport

   - Attributes: reportDate (*string*), summary (*string*), recommendations (*string*)

   - Methods: generateSummary(*user:User*), generateRecommendations (*user:User*)

7. Suggestion
    - Attributes: suggestionText (*string*)
    - Methods: provideSuggestion(*user: User*)
8. ImprovementPlan
    - Attributes: planDetails(*string*)
    - Methods: createPlan (*user: User*)

(ii) <u>Relationships</u>

<u>Inheritance</u>

- Classes that involve inheritance relationship are *class Suggestion* which is considered as a base class and *class ImprovementPlan* which is considered as a derived class.

- The reason that these two classes are in inheritance relationship is because a *class Suggestion* represents a basic recommendation or piece of advice. Therefore, it could be a general suggestion to improve sleep quality. In the other hand, a *class ImprovementPlan* is a more comprehensive and detailed plan that might include multiple suggestions. It is inherited from *class Suggestion* because it can be seen as an extension of *class Suggestion* but with additional details of method such as 'createPlan' method.

- Besides, the reason they should be modelled with inheritance relationship is because inheritance leads *class ImprovementPlan*, a derived class can also use the attributes of *class Suggestion*, a base class and on top of that, we can add more functionalities by modifying the base *class Suggestion* so that a detailed series of suggestions aimed at improving sleep quality can be generated.

Aggregation

- Definition: a special type of association which is one way relationship. An aggregation relationship is implemented by objects containing pointers to other objects.

- The reason for using aggregation is the child objects can exist independently of the parents' object. This means that the lifecycle of the child is not tied to the lifecycle of the parents' class. Second, since the child's object is not tightly coupled with the parents' class, they can be reused in different contexts or with different parents' objects. Third, aggregation allows for building modular systems where components can be easily swapped or replaced without affecting the rest of the coding. Finally, aggregation simplifies maintenance and updates because changes to the part do not necessarily impact the parents.

- *Class sleepQualityAssessment* has a collection of pointers to *class SleepData* objects *sleepData: vector<SleepData*>*. This indicates aggregation because *class SleepData* object can exist independently of the *class SleepQualityAssessment* object that references them. They are aggregated into SleepQualityAssessment to analyze and assess sleep quality, but their lifecycle is not tied to it.

- User class has a collection of pointers to *class SleepQualityAssessment* objects *SleepQualityAssessment: SleepQualityAssessment**. This indicates aggregation because *class SleepQualityAssessment* object can exist independently of the *class User* object that references them. It can be associated with multiple users and its lifecycle is not managed by the *class User*.

- *User* class has a collection of pointers to *class Suggestion* objects *suggestions: vector<Suggestion*>*. This indicates aggregation because *class Suggestion* object can exist independently and be shared among different users.

- *User* class has a collection of pointers to *class SleepReport* objects *SleepReport: SleepReport**. This indicates aggregation because *class SleepReport* object can exist independently of the *class User*.

Composition

- Definition: a restricted version of aggregation in which the enclosing and enclosed objects are highly dependent on each other.

- The reason for using composition is that the parents object is responsible for the lifecycle or the child object. The child cannot exist without the parents' class. Second, composition enhances encapsulation by ensuring that the child is tightly bound to the parents' class, making the parents object a single cohesive unit. Third, the child's objects are created, managed, and destroyed by the parents' object, ensuring consistency and integrity withing the parents' class.

- The *SleepSession* class has a collection of *class SleepData* objects *SleepData: vector<SleepData>*. This relationship is composition because *class SleepData* objects are part of a *SleepSession* class, and their lifecycle is tied to the *class SleepSession*.

- The *class User* has a collection of *class SleepSession* objects *sleepSession: vector<SleepSession>*. This suggests composition because the *SleepSession* objects are tightly coupled with the *User* class and their lifecycle is managed and controlled by the *User* class.

- The *User* class has a *class SleepCycleAnalyzer* object *SleepCycleAnalyzer: SleepCycleAnalyzer*. This is a composition relationship because the *class SleepCycleAnalyzer* is a part of the *User* class, and its lifecycle is managed by the *User* class. It is created and destroyed with the *class User.*

Section C: Class Diagram

The class diagram represents a sleep analysis system that collects sleep data, analyzes it, and provides personalized recommendations to improve sleep quality.

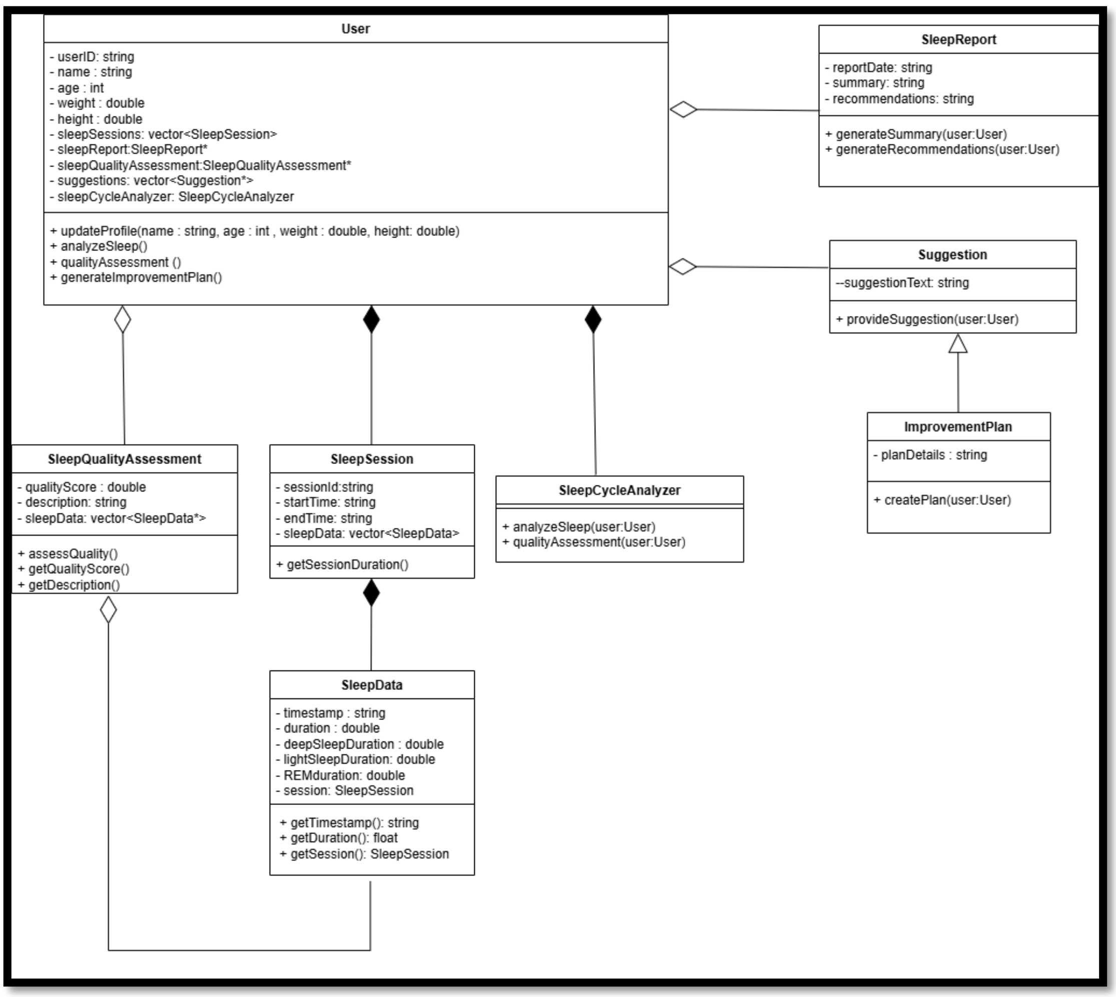

Figure 2: Class diagram of Sleep Analyzer System