Software (API) Documentation

Prepared by:

Arwa Almrzoqi

441212001

Hissah Almousa

441211996

Munirah Aldaosari

441211997

Qassim University
Software Engineering and Knowledge Engineering
CSC606.

Instructor:
Dr. Faisal Alhwikem.

17/2/2023

Table of Contents

1. CODE IMPLEMENTATION 1

2. DESIGN PATTERNS 3

1. CODE IMPLEMENTATION

In this section, a code implementation (API) of the Academic Advisor software is presented below. The code implemented using Python for the components and classes, including methods and attributes.

```
class Student:
  def __init__(self, student_id, major, department):
    self.student_id = student_id
    self.major = major
    self.department = department
    self.reservations = []  # list of reservations made by the student

  def login(self, id, password):
    # Code implementation for student login

  def make_reservation(self, reservation_number, date, time):
        # Code implementation for student to make a reservation
    return reservation

  def check_reservation_status(self, reservation_number):
            # Code implementation for student to check reservation status
        return reservation.status
    #return 'not found'


class Advisor:
  def __init__(self, advisor_id, department):
    self.advisor_id = advisor_id
    self.department = department
    self.schedule = Schedule(advisor_id)
    self.reservations = []  # list of reservations made with this advisor

  def login(self, id, password):
    # Code implementation for advisor login

  def view_schedule(self):
    return self.schedule.view_schedule()
    def manage_schedule(self):
        self.schedule.manage_schedule()

    def confirm_appointment(self, reservation_number):
                  # Code implementation for advisor to confirm appointment
          reservation.change_status('confirmed')

    def reject_appointment(self, reservation_number):
                  # Code implementation for advisor to reject appointment
          reservation.change_status('rejected')

    def get_reservation(self, reservation_number):
                   # Code implementation for advisor to get the reservation number from the
list
            return reservation
        #return None


class Reservations:
    def __init__(self, reservation_number, date, time, status):
        self.reservation_number = reservation_number
        self.date = date
        self.time = time
        self.status = status

    def change_status(self, new_status):
        self.status = new_status


class Schedule:
    def __init__(self, advisor_id):
        self.advisor_id = advisor_id
        self.schedule = {}  # List of date-time slots with availability status

    def add_schedule(self, date, time):
        key = f'{date}-{time}'
        if key not in self.schedule:
            self.schedule[key] = 'available'

    def update_schedule(self, date, time, new_status):
        key = f'{date}-{time}'
        if key in self.schedule:
            self.schedule[key] = new_status
```

```
def view_schedule(self):
    return self.schedule

def manage_schedule(self):
    # Code implementation for advisor to manage their schedule
    pass
```

---

2. DESIGN PATTERNS

Two design patterns have been implemented in this software:

- The first pattern is ‘Observer’ which is behavioral pattern that allows objects to subscribe to and receive updates from a subject. In the context of our academic advisor software, the Observer pattern has been used to allow the student and advisor to be involved to the schedule class, so that they can be notified of any changes to the schedule. The code implementation of ‘observer’ class is as follows:

```
class Observer:
    def update(self):
        pass

class Student(Observer):
    def update(self):
        print("Schedule updated")

class Advisor(Observer):
    def update(self):
        print("Schedule updated")

class Schedule:
    def __init__(self):
        self.observers = []

    def add_observer(self, observer):
        self.observers.append(observer)

    def remove_observer(self, observer):
        self.observers.remove(observer)

    def notify_observers(self):
        for observer in self.observers:
            observer.update()
    def update_schedule(self):
        # code implementation to update schedule
        self.notify_observers()
```

- The second pattern is the Factory Method pattern which is a creational pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. In the context of our academic advisor software, the Factory Method pattern has been used for ‘Person’ class to allow for the creation of either Student or Advisor, depending on the login credentials provided.
  The code implementation of ‘Person’ class is as follows:

```
from abc import ABC, abstractmethod
     # abstract method is used to ensure that it's implemented in each subclass

class Person(ABC):
    @abstractmethod
    def login(self, ID, Password):
        pass

class Student(Person):
    def login(self, ID, Password):
        # implementation to check student login credentials
        pass

class Advisor(Person):
    def login(self, ID, Password):
        # implementation to check advisor login credentials
        pass

class PersonFactory:

    def create_person(ID, Password):
        if is_student(ID, Password):
            return Student()
        elif is_advisor(ID, Password):
            return Advisor()
        else:
            raise ValueError("Invalid login credentials")
```