

README:

Medical Center

Table of contents

- Introduction
- Features
- Technologies
- The logic behind classes and method
- Testing

Introduction

'Medical center' is a piece of code developed for a university project. It has been written based on a UML design. It is going to check students' understanding of object-oriented Python. The code can be extended and turned into a full-fledged application with different access for different people including healthcare professionals, receptionists, and patients. They can access to various information. Considering access level, doctors can carry out consultation as well as issuing prescriptions for patients; Receptionists use the system to make/cancel appointments, add new appointments or find the next ones; Patients can request appointments as well as request repeat prescriptions.

Features

'Medical Center' possesses some features:

- ✓ It is a piece of code with no GUI interface and no connection to database
- ✓ It is executed using command line or terminal
- ✓ It can add new appointment to the existing list
- ✓ It can delete an appointment from the existing list

Technologies

'Medical Center' has been written in Visual Studio Code editor and then transferred to *Codio* which is an ubuntu-based editor. To develop the code, *Python* programming language and some of its libraries such as *Pandas* have been used. The *Pandas* has been applied to display the output in a table style. For the testing purpose, *Unittest* has been used.

'Medical Center' can be run in devices with different operating systems when extended and turned into an application.

The logic behind classes and methods

Looking at the design plan, relationships of composition, aggregation and association are seen which among them, the composition is the dominant. To develop the code, I started with *Appointment* class which every other class was directly or indirectly related to. To write *Appointment* class, two other classes - *HealthcareProfessional* and *Patient* - were needed. After creating those two, it was

time to put them as attributes in *Appointment* class. Back to *HealthcareProfessional* class, we can see the class is the parent of two other classes - *Doctor* and *Nurse*. Actually, *Doctor* and *Nurse* classes inherit all attributes and a method called 'carryout_consultation()' from *HealthcareProfessional* class. To show this inheritance, `super()` has been used. However, one of the children (*Doctor*) has an extra method called 'Issue_prescription()'. Back to *Patient* class, this class has 3 attributes itself but other classes - *Appointment*, *HealthcareProfessional* and *Prescription* - have been added to it as new attributes; Using added attributes, we can define methods which can be called later.

Receptionist like *Patient* class has other classes as its attributes which can be used later in method callings. Among the classes, *AppointmentSchedule* is a bit tricky but the most important one which can be used as a database of our future app.

AppointmentSchedule class has a list of dictionaries (keys and values). To create a new dictionary, attributes of other classes have been assigned as values to the keys. Now, it is time to add the newly created dictionary to the list using 'add_appointment' method and display it in a table style format using 'pandas.DataFrame()'. Another method defined in *AppointmentSchedule* class is called 'cancel_appointment' which tries to find an item index and remove the item using the index. The last method - 'find_next_appointment' - has been defined to find the closest date to today's date.

Testing

Testing is a process to evaluate the code to see whether it does what is supposed to do. For 'Medical Center', *unit testing* has been applied. In unit testing, small part of the code - unit - will be evaluated and it can be done in automated or manual form.

For 'Medical Center' scenario, some methods of the classes have been tested using *unittest*.

The screenshot shows a Visual Studio Code editor with a file named `test_appointment.py`. The code defines a `TestCarryoutConsultation` class that inherits from `unittest.TestCase`. It includes a test method `test_carryout_consultation` that creates an `Appointment` object and a `Doctor` object, then calls the `carryout_consultation` method. The terminal output shows the execution of the test, which passes. The output also displays a list of appointments and the next available appointment date.

```
1 import unittest
2 from appointment import Prescription, HealthcareProfessional, Doctor, Nurse, Appoi
3
4 class TestCarryoutConsultation(unittest.TestCase):
5     def test_carryout_consultation(self):
6         appointment = Appointment('Emergency', 'Dr. Hudson', 'Patient Sara')
7         doctor = Doctor('Dr. Clifford', 129, 'Patient Vance', 'Capsules', 6, 45)
```

Dr. Richardson with employee number 222 carries out a consultation for Patient Nina.
Dr. Dayton with employee number 112 carries out a consultation for Patient Paul.
Dr. Dayton with employee number 112 issues a medication for Patient Paul which includes 3 ointments 35 mg.
Nurse simpson with employee number 223 carries out a consultation for Patient Linda.
Heather with employee number 432 arranges a Standard appointment for Patient Simon with Nurse Harrison.
Heather with employee number 432 cancels the appointment of Patient Simon with Nurse Harrison.

Appointment_Type	Patient	Healthcare_Professional	Healthcare_professional
Standard	Jane Joyce	Dr. Jones	NaN
Emergency	Tina Tan	Nurse Dawson	NaN
Standard	George Kidman	NaN	Dr. Swindon

Appointment_Type Patient Healthcare_Professional
0 Emergency Tina Tan Nurse Dawson

Next Available Appointment: 2022-06-21
.....
Ran 6 tests in 0.002s
OK

Figure 1. Testing result of 'Medical Center'

Python codes

The screenshot shows a Visual Studio Code editor with a file named `test_oois_systemImpl...`. The code defines a `HealthcareProfessional` class and a `Doctor` class that inherits from it. The `HealthcareProfessional` class has an `__init__` method that takes `hname`, `hnumber`, and `patient` as arguments. The `Doctor` class has an `__init__` method that takes `hname`, `hnumber`, `patient`, `type`, `quantity`, and `dosage` as arguments. The `carryout_consultation` method of the `HealthcareProfessional` class returns a string describing the consultation.

```
1 import pandas
2 import datetime
3
4 class HealthcareProfessional:
5     """Represents a healthcare professional like doctor or nurse."""
6     def __init__(self, hname, hnumber, patient):
7         self.hname = hname
8         self.hnumber = hnumber
9         # Using other class as its attribute
10        self.patient = patient
11
12    def carryout_consultation(self):
13        return f'{self.hname} with employee number {self.hnumber} carries out a co
14
15
16 class Doctor(HealthcareProfessional):
17     """Represents a doctor of healthcare professionals"""
18    def __init__(self, hname, hnumber, patient, type, quantity, dosage):
19        # Accesses the properties and methods of the parent using super()
20        super().__init__(hname, hnumber, patient)
21        # Using other classes as its attributes
```

```

22     self.type = type
23     self.quantity = quantity
24     self.dosage = dosage
25     super().carryout_consultation()
26
27     def issue_prescription(self):
28         return f'{self.hname} with employee number {self.hnumber} issues a medicat
29
30
31 class Nurse(HealthcareProfessional):
32     """Represents a nurse of healthcare professionals"""
33     def __init__(self, hname, hnumber, patient):
34         # Accesses the properties and methods of the parent using super()
35         super().__init__(hname, hnumber, patient)
36         super().carryout_consultation()
37
38
39 class Appointment:
40     """Represents an appointment"""
41     def __init__(self, type, healthcare_professional, patient):
42         # Using other classes as its attributes

```

```

43     self.type = type
44     self.healthcare_professional = healthcare_professional
45     self.patient = patient
46
47
48 class Prescription:
49     """Represents a prescription"""
50     def __init__(self, type, quantity, dosage, doctor, patient):
51         # Using other classes as its attributes
52         self.type = type
53         self.quantity = quantity
54         self.dosage = dosage
55         self.doctor = doctor
56         self.patient = patient
57
58
59 class Patient:
60     """Represents a patient"""
61     def __init__(self, name, address, phone, appointment_type, healthcare_professi
62         self.name = name
63         self.address = address

```

```

64     self.phone = phone
65     # Using other classes as its attributes
66     self.appointment_type = appointment_type
67     self.healthcare_professional = healthcare_professional
68     self.prescription_type = prescription_type
69     self.dosage = dosage
70
71     def request_appointment(self):
72         return f"{self.name} living in {self.address} with phone number {self.phone} requests {self.appointment_type} with dosage {self.dosage}"
73
74     def request_repeat_prescription(self):
75         return f"{self.name} requests {self.healthcare_professional[0]} with employee number {self.employee_number} for {self.prescription_type} with dosage {self.dosage}"
76
77
78 class Receptionist:
79     """Represents a receptionist"""
80     def __init__(self, name, employee_number, appointment_type, healthcare_professional):
81         self.name = name
82         self.employee_number = employee_number
83         # Using other classes as its attributes
84         self.appointment_type = appointment_type

```

```

85         self.healthcare_professional = healthcare_professional
86         self.patient = patient
87
88     def make_appointment(self):
89         return f"{self.name} with employee number {self.employee_number} arranges {self.appointment_type} with dosage {self.dosage} for {self.patient}"
90
91     def cancel_appointment(self):
92         return f"{self.name} with employee number {self.employee_number} cancels {self.appointment_type} with dosage {self.dosage} for {self.patient}"
93
94
95
96 class AppointmentSchedule:
97     """Represents an appointment schedule"""
98     def __init__(self, type, healthcare_professional, patient):
99         # a list of dictionaries
100         self.appointments = [
101             {"Appointment_Type": "Standard", "Patient": "Jane Joyce", "Healthcare_Professional": "Dr. Jane Joyce", "Dosage": "100mg"},
102             {"Appointment_Type": "Emergency", "Patient": "Tina Tan", "Healthcare_Professional": "Dr. Tina Tan", "Dosage": "50mg"}
103         ]
104
105         # an empty dictionary

```

```

106 self.new_dictionary = {}
107 # assigning attributes of other classes as values of the dictionary's keys
108 self.new_dictionary["Appointment_Type"] = type
109 self.new_dictionary["Healthcare_professional"] = healthcare_professional
110 self.new_dictionary["Patient"] = patient
111
112 def add_appointment(self):
113     "Adds the newly made dictionary to the list"
114     self.appointments.append(self.new_dictionary)
115     # calling display and display the data in table style
116     return self.display()
117
118 def cancel_appointment(self, appointment):
119     for item in self.appointments:
120         for itemvalue in item.values():
121             if itemvalue == appointment:
122                 # returns the position of a value in a list
123                 index = self.appointments.index(item)
124                 # removes an item in a list
125                 del self.appointments[index]
126                 return self.display()

```

```

128 def find_next_appointment(self, date_strings, date_format):
129     "Finds the closest date to today's date"
130     # returns today date in string format of YYYY-MM-DD
131     today = datetime.date.today()
132     # turns string format into object format
133     date_object = [datetime.datetime.strptime(date_string, date_format).date()
134                     for date_string in date_strings]
135     result = [date for date in date_object
136              if date >= today]
137     # returns the smallest and closest to today's date
138     return "Next Available Appointment: " + str(min(result))
139
140 def __str__(self):
141     return f"Name: {self.patient}, Appointment: {self.type}, Healthcare profes
142
143 def display(self):
144     "Represents the data in table style format"
145     return pandas.DataFrame(self.appointments)
146
147

```

```
149
150
151 patient = Patient("Patient Jane", "Leeds", "989145", "emergency", ["Dr. Johnson",
152 print(patient.request_appointment())
153 print(patient.request_repeat_prescription())
154
155 appointment = Appointment("Standard", ("Nurse Josephine", "221"), ("Patient Alec",
156 appointment.healthcare_professional = HealthcareProfessional("Dr. Richardson", "22
157 print(appointment.healthcare_professional.carryout_consultation())
158
159 doctor = Doctor("Dr. Dayton", "112", "Patient Paul", "ointments", 3, 35)
160 print(doctor.carryout_consultation())
161 print(doctor.issue_prescription())
162
163 nurse = Nurse("Nurse simpson", "223", "Patient Linda")
164 print(nurse.carryout_consultation())
165
166 receptionist = Receptionist("Heather", 432, "Standard", "Nurse Harrison", "Patient
167 print(receptionist.make_appointment())
168 print(receptionist.cancel_appointment())
169
```

```
169
170 print("")
171 appointmentschedule1 = AppointmentSchedule("Standard", "Dr. Swindon", "George Kidm
172 appointmentschedule1.add_appointment()
173 print(appointmentschedule1.display())
174
175 print("")
176 appointmentschedule2 = AppointmentSchedule("Standard", "Dr. Swindon", "George Kidm
177 appointmentschedule2.cancel_appointment("Dr. Jones")
178 print(appointmentschedule2.display())
179
180 print("")
181 print(appointmentschedule2.find_next_appointment(["2022-08-21", "2022-07-21", "202
182
183 |
```


Testing



```
1 import unittest
2 from oois_systemImplementation import Prescription, HealthcareProfessional, Doctor
3
4 class TestCarryoutConsultation(unittest.TestCase):
5     def test_carryout_consultation(self):
6         appointment = Appointment('Emergency', 'Dr. Hudson', 'Patient Sara')
7         doctor = Doctor('Dr. Clifford', 129, 'Patient Vance', 'Capsules', 6, 45)
8         nurse = Nurse('Nurse Hillton', 234, 'Patient Harvey')
9         appointment.healthcare_professional = HealthcareProfessional('Dr. Hudson',
10 self.assertTrue(doctor.carryout_consultation(), True)
11 self.assertTrue(nurse.carryout_consultation(), True)
12
13
14 class TestRequestAppointment(unittest.TestCase):
15     def test_request_appointment(self):
16         patient = Patient('Patient Jean', 'Brighton', '765432', 'Standard', 'Nurse
17 self.assertTrue(patient.request_appointment(), True)
18
19 class TestRequestRepeatPrescription(unittest.TestCase):
20     def test_cancel_appointment(self):
21         patient = Patient('Patient Jean', 'Brighton', '765432', 'Standard', 'Nu
```



```
22 self.assertTrue(patient.request_repeat_prescription(), True)
23
24
25 class TestMakeAppointment(unittest.TestCase):
26     def test_make_appointment(self):
27         receptionist = Receptionist('Joana', 456, 'Emergency', 'Nurse Pendleton',
28 self.assertTrue(receptionist.make_appointment(), True)
29
30 class TestCancelAppointment(unittest.TestCase):
31     def test_cancel_appointment(self):
32         receptionist = Receptionist('Joana', 456, 'Emergency', 'Nurse Pendleton',
33 self.assertTrue(receptionist.cancel_appointment(), True)
34
35
36 class TestAddAppointment(unittest.TestCase):
37     def test_add_appointment(self):
38         appointmentschedule = AppointmentSchedule("Standard", "Dr. Swindon", "Geor
39 assert appointmentschedule is not None
40
41 class TestCancelAppointment(unittest.TestCase):
42     def test_cancel_appointment(self):
```

