|  |  |
| --- | --- |
|  | Министерство науки и высшего образования Российской Федерации  Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»  (МГТУ им. Н.Э. Баумана) |
| ФАКУЛЬТЕТ «ИНЖЕНЕРНЫЙ БИЗНЕС И МЕНЕДЖМЕНТ»  КАФЕДРА «ПРОМЫШЛЕННАЯ ЛОГИСТИКА» (ИБМ-3)  Лабораторная работа №5  По дисциплине:  «Парадигмы и конструкции языков программирования»  Студент ИБМ3- 34Б П. Я. Головастикова  (Подпись, дата)    Руководитель Ю. Е. Гапанюк  (Подпись, дата)  2025 г. | |

За основу я взяла предметную область «Поликлиника», реализовав ее в трех паттернах.

Среди структурных паттернов, выбрала «Декоратор». Код программы:

from dataclasses import dataclass  
from datetime import datetime  
from typing import List, Optional  
  
@dataclass  
class Patient:  
 id: int  
 name: str  
 age: int  
 medical\_records: List["MedicalRecord"] = None  
  
 def \_\_post\_init\_\_(self):  
 if self.medical\_records is None:  
 self.medical\_records = []  
  
@dataclass  
class Doctor:  
 id: int  
 name: str  
 specialization: str  
 department: "Department"  
  
@dataclass  
class Department:  
 id: int  
 name: str  
 doctors: List[Doctor] = None  
  
 def \_\_post\_init\_\_(self):  
 if self.doctors is None:  
 self.doctors = []  
  
@dataclass  
class Appointment:  
 id: int  
 patient: Patient  
 doctor: Doctor  
 time: datetime  
  
@dataclass  
class MedicalRecord:  
 id: int  
 patient: Patient  
 diagnosis: str  
 treatment: str  
 date: datetime  
  
class PolyclinicFacade:  
 def \_\_init\_\_(self):  
 self.\_patients = []  
 self.\_doctors = []  
 self.\_departments = []  
 self.\_appointments = []  
 self.\_medical\_records = []  
  
 def add\_patient(self, name: str, age: int) -> Patient:  
 patient\_id = len(self.\_patients) + 1  
 patient = Patient(id=patient\_id, name=name, age=age)  
 self.\_patients.append(patient)  
 return patient  
  
 def add\_doctor(self, name: str, specialization: str, department\_name: str) -> Doctor:  
 department = self.\_find\_or\_create\_department(department\_name)  
 doctor\_id = len(self.\_doctors) + 1  
 doctor = Doctor(id=doctor\_id, name=name, specialization=specialization, department=department)  
 self.\_doctors.append(doctor)  
 department.doctors.append(doctor)  
 return doctor  
  
 def create\_appointment(self, patient\_id: int, doctor\_id: int, time: datetime) -> Appointment:  
 patient = self.\_find\_patient(patient\_id)  
 doctor = self.\_find\_doctor(doctor\_id)  
 appointment\_id = len(self.\_appointments) + 1  
 appointment = Appointment(id=appointment\_id, patient=patient, doctor=doctor, time=time)  
 self.\_appointments.append(appointment)  
 return appointment  
  
 def add\_medical\_record(self, patient\_id: int, diagnosis: str, treatment: str) -> MedicalRecord:  
 patient = self.\_find\_patient(patient\_id)  
 record\_id = len(self.\_medical\_records) + 1  
 record = MedicalRecord(  
 id=record\_id,  
 patient=patient,  
 diagnosis=diagnosis,  
 treatment=treatment,  
 date=datetime.now()  
 )  
 patient.medical\_records.append(record)  
 self.\_medical\_records.append(record)  
 return record  
  
 def \_find\_patient(self, patient\_id: int) -> Patient:  
 for patient in self.\_patients:  
 if patient.id == patient\_id:  
 return patient  
 raise ValueError("Пациент не найден")  
  
 def \_find\_doctor(self, doctor\_id: int) -> Doctor:  
 for doctor in self.\_doctors:  
 if doctor.id == doctor\_id:  
 return doctor  
 raise ValueError("Доктор не найден")  
  
 def \_find\_or\_create\_department(self, name: str) -> Department:  
 for department in self.\_departments:  
 if department.name == name:  
 return department  
 department\_id = len(self.\_departments) + 1  
 new\_department = Department(id=department\_id, name=name)  
 self.\_departments.append(new\_department)  
 return new\_department  
  
 def get\_patient\_appointments(self, patient\_id: int) -> List[Appointment]:  
 return [app for app in self.\_appointments if app.patient.id == patient\_id]  
  
 def get\_doctor\_appointments(self, doctor\_id: int) -> List[Appointment]:  
 return [app for app in self.\_appointments if app.doctor.id == doctor\_id]

Результат выполнения программы был выдан с помощью трех тестов: TDD, BDD, создание Mock-объектов.

TDD фреймворк с добавлением mock-объектов:

import unittest  
from unittest.mock import Mock  
import sys  
import os  
  
sys.path.append(os.path.dirname(os.path.abspath(\_\_file\_\_)))  
  
from polyclinic import \*  
  
  
class TestPolyclinicTDD(unittest.TestCase):  
  
 def setUp(self):  
 self.polyclinic = PolyclinicFacade()  
 self.patient = self.polyclinic.add\_patient("Иван Иванов", 30)  
 self.doctor = self.polyclinic.add\_doctor("Доктор Смирнов", "Терапевт", "Терапия")  
  
 def test\_add\_patient(self):  
 patient = self.polyclinic.add\_patient("Петр Петров", 25)  
 self.assertEqual(patient.name, "Петр Петров")  
 self.assertEqual(patient.age, 25)  
 self.assertEqual(patient.id, 2)  
 self.assertEqual(len(patient.medical\_records), 0)  
  
 def test\_add\_doctor(self):  
 doctor = self.polyclinic.add\_doctor("Доктор Орлов", "Хирург", "Хирургия")  
 self.assertEqual(doctor.name, "Доктор Орлов")  
 self.assertEqual(doctor.specialization, "Хирург")  
 self.assertEqual(doctor.department.name, "Хирургия")  
  
 def test\_create\_appointment(self):  
 appointment\_time = datetime(2024, 5, 20, 10, 0)  
 appointment = self.polyclinic.create\_appointment(  
 self.patient.id, self.doctor.id, appointment\_time  
 )  
 self.assertEqual(appointment.patient.id, self.patient.id)  
 self.assertEqual(appointment.doctor.id, self.doctor.id)  
 self.assertEqual(appointment.time, appointment\_time)  
  
 def test\_add\_medical\_record(self):  
 record = self.polyclinic.add\_medical\_record(  
 self.patient.id, "Грипп", "Постельный режим"  
 )  
  
 self.assertEqual(record.patient.id, self.patient.id)  
 self.assertEqual(record.diagnosis, "Грипп")  
 self.assertEqual(record.treatment, "Постельный режим")  
 self.assertIn(record, self.patient.medical\_records)  
  
  
 def test\_mock\_patient\_object(self):  
 mock\_patient = Mock(spec=Patient)  
 mock\_patient.id = 1  
 mock\_patient.name = "Mock Patient"  
 mock\_patient.age = 40  
 mock\_patient.medical\_records = []  
  
 result = mock\_patient.name  
  
 self.assertEqual(result, "Mock Patient")  
 self.assertEqual(mock\_patient.id, 1)  
  
 def test\_mock\_appointment\_creation(self):  
  
 mock\_patient = Mock(spec=Patient)  
 mock\_doctor = Mock(spec=Doctor)  
 mock\_time = datetime(2024, 5, 20, 10, 0)  
  
 mock\_patient.id = 1  
 mock\_doctor.id = 1  
  
 appointment = Appointment(  
 id=1,  
 patient=mock\_patient,  
 doctor=mock\_doctor,  
 time=mock\_time  
 )  
  
 self.assertEqual(appointment.patient.id, 1)  
 self.assertEqual(appointment.doctor.id, 1)  
 self.assertEqual(appointment.time, mock\_time)  
  
 def test\_department\_creation(self):  
 doctor = self.polyclinic.add\_doctor("Доктор Новый", "Кардиолог", "Кардиология")  
  
 self.assertEqual(doctor.department.name, "Кардиология")  
 self.assertIn(doctor, doctor.department.doctors)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

Результат работы программы:

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

Проверка с помощью BDD-фреймворка:

Сценарии для проверки:

Feature: Управление поликлиникой  
 Как администратор поликлиники  
 Я хочу управлять пациентами, докторами и записями  
 Чтобы обеспечить эффективную работу медицинского учреждения  
  
 Scenario: Добавление нового пациента  
 Given поликлиника инициализирована  
 When я добавляю пациента "Мария Сидорова" возрастом 28 лет  
 Then пациент "Мария Сидорова" должен быть в системе  
 And у пациента должен быть ID  
 And возраст пациента должен быть 28 лет  
  
 Scenario: Добавление доктора в отделение  
 Given поликлиника инициализирована  
 When я добавляю доктора "Доктор Белов" со специализацией "Невролог" в отделение "Неврология"  
 Then доктор "Доктор Белов" должен быть в системе  
 And доктор должен иметь специализацию "Невролог"  
 And доктор должен быть привязан к отделению "Неврология"  
  
 Scenario: Создание записи на прием  
 Given поликлиника инициализирована  
 And есть пациент "Сергей Козлов" возрастом 35 лет  
 And есть доктор "Доктор Ветров" в отделении "Терапия"  
 When я создаю запись пациента к доктору на "2024-05-20 10:00"  
 Then запись должна быть создана успешно  
 And запись должна содержать правильного пациента  
 And запись должна содержать правильного доктора  
  
 Scenario: Добавление медицинской записи  
 Given поликлиника инициализирована  
 And есть пациент "Ольга Новикова" возрастом 29 лет  
 When я добавляю медицинскую запись с диагнозом "Мигрень" и лечением "Отдых"  
 Then медицинская запись должна быть добавлена в историю пациента  
 And запись должна содержать диагноз "Мигрень"  
 And запись должна содержать лечение "Отдых"

Программа, проверяющая сценарии:

import sys  
import os  
  
sys.path.append(os.path.join(os.path.dirname(\_\_file\_\_), '..', '..'))  
  
from behave import given, when, then  
from datetime import datetime  
from polyclinic import \*  
  
@given('поликлиника инициализирована')  
def step\_impl(context):  
 context.polyclinic = PolyclinicFacade()  
  
@given('есть пациент "{name}" возрастом {age:d} лет')  
def step\_impl(context, name, age):  
 context.patient = context.polyclinic.add\_patient(name, age)  
  
@given('есть доктор "{doctor\_name}" в отделении "{department}"')  
def step\_impl(context, doctor\_name, department):  
 context.doctor = context.polyclinic.add\_doctor(doctor\_name, "Специалист", department)  
  
@when('я добавляю пациента "{name}" возрастом {age:d} лет')  
def step\_impl(context, name, age):  
 context.patient = context.polyclinic.add\_patient(name, age)  
  
@when('я добавляю доктора "{doctor\_name}" со специализацией "{specialization}" в отделение "{department}"')  
def step\_impl(context, doctor\_name, specialization, department):  
 context.doctor = context.polyclinic.add\_doctor(doctor\_name, specialization, department)  
  
@when('я создаю запись пациента к доктору на "{appointment\_time}"')  
def step\_impl(context, appointment\_time):  
 time\_obj = datetime.strptime(appointment\_time, "%Y-%m-%d %H:%M")  
 context.appointment = context.polyclinic.create\_appointment(  
 context.patient.id, context.doctor.id, time\_obj  
 )  
  
@when('я добавляю медицинскую запись с диагнозом "{diagnosis}" и лечением "{treatment}"')  
def step\_impl(context, diagnosis, treatment):  
 context.medical\_record = context.polyclinic.add\_medical\_record(  
 context.patient.id, diagnosis, treatment  
 )  
  
@then('пациент "{name}" должен быть в системе')  
def step\_impl(context, name):  
 assert context.patient.name == name  
  
@then('у пациента должен быть ID')  
def step\_impl(context):  
 assert context.patient.id is not None  
 assert context.patient.id > 0  
  
@then('возраст пациента должен быть {age:d} лет')  
def step\_impl(context, age):  
 assert context.patient.age == age  
  
@then('доктор "{name}" должен быть в системе')  
def step\_impl(context, name):  
 assert context.doctor.name == name  
  
@then('доктор должен иметь специализацию "{specialization}"')  
def step\_impl(context, specialization):  
 assert context.doctor.specialization == specialization  
  
@then('доктор должен быть привязан к отделению "{department}"')  
def step\_impl(context, department):  
 assert context.doctor.department.name == department  
  
@then('запись должна быть создана успешно')  
def step\_impl(context):  
 assert context.appointment is not None  
 assert context.appointment.id > 0  
  
@then('запись должна содержать правильного пациента')  
def step\_impl(context):  
 assert context.appointment.patient.id == context.patient.id  
  
@then('запись должна содержать правильного доктора')  
def step\_impl(context):  
 assert context.appointment.doctor.id == context.doctor.id  
  
@then('медицинская запись должна быть добавлена в историю пациента')  
def step\_impl(context):  
 assert context.medical\_record in context.patient.medical\_records  
  
@then('запись должна содержать диагноз "{diagnosis}"')  
def step\_impl(context, diagnosis):  
 assert context.medical\_record.diagnosis == diagnosis  
  
@then('запись должна содержать лечение "{treatment}"')  
def step\_impl(context, treatment):  
 assert context.medical\_record.treatment == treatment

Результат выполнения:

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Из порождающих паттернов был реализован «Фабричный метод». Код программы:

from dataclasses import dataclass  
from datetime import datetime  
from typing import List, Optional  
from abc import ABC, abstractmethod  
from enum import Enum  
  
class DepartmentType(Enum):  
 THERAPY = "Терапия"  
 SURGERY = "Хирургия"  
 CARDIOLOGY = "Кардиология"  
 NEUROLOGY = "Неврология"  
  
  
class DepartmentFactory(ABC):  
 @abstractmethod  
 def create\_department(self, name: str) -> 'Department':  
 pass  
  
 @abstractmethod  
 def create\_doctor(self, name: str, specialization: str) -> 'Doctor':  
 pass  
  
  
class TherapyDepartmentFactory(DepartmentFactory):  
 def create\_department(self, name: str) -> 'Department':  
 return Department(id=1, name=name, department\_type=DepartmentType.THERAPY)  
  
 def create\_doctor(self, name: str, specialization: str) -> 'Doctor':  
 return Doctor(id=1, name=name, specialization=specialization, department=None)  
  
  
class SurgeryDepartmentFactory(DepartmentFactory):  
 def create\_department(self, name: str) -> 'Department':  
 return Department(id=1, name=name, department\_type=DepartmentType.SURGERY)  
  
 def create\_doctor(self, name: str, specialization: str) -> 'Doctor':  
 return Doctor(id=1, name=name, specialization=specialization, department=None)  
  
  
class DepartmentFactoryCreator:  
 @staticmethod  
 def get\_factory(dept\_type: DepartmentType) -> DepartmentFactory:  
 factories = {  
 DepartmentType.THERAPY: TherapyDepartmentFactory(),  
 DepartmentType.SURGERY: SurgeryDepartmentFactory(),  
 DepartmentType.CARDIOLOGY: TherapyDepartmentFactory(),   
 DepartmentType.NEUROLOGY: TherapyDepartmentFactory()   
 }  
 return factories[dept\_type]

Из поведенческих паттернов был выполнен «Наблюдатель». Код программы:

class AppointmentObserver(ABC):  
 @abstractmethod  
 def update(self, appointment: 'Appointment', event\_type: str) -> None:  
 pass  
  
  
class EmailNotificationObserver(AppointmentObserver):  
 def update(self, appointment: 'Appointment', event\_type: str) -> None:  
 if event\_type == "created":  
 print(f"📧 Email: Создана запись на прием для {appointment.patient.name} "  
 f"к доктору {appointment.doctor.name} на {appointment.time}")  
 elif event\_type == "cancelled":  
 print(f"📧 Email: Отменена запись на прием для {appointment.patient.name}")  
  
  
class SMSNotificationObserver(AppointmentObserver):  
 def update(self, appointment: 'Appointment', event\_type: str) -> None:  
 if event\_type == "created":  
 print(f"📱 SMS: Новая запись к {appointment.doctor.name} на {appointment.time.strftime('%H:%M')}")  
 elif event\_type == "cancelled":  
 print(f"📱 SMS: Запись отменена")  
  
  
class ScheduleManagerObserver(AppointmentObserver):  
 def update(self, appointment: 'Appointment', event\_type: str) -> None:  
 if event\_type == "created":  
 print(f"📅 Schedule: Добавлена запись в расписание доктора {appointment.doctor.name}")  
 elif event\_type == "cancelled":  
 print(f"📅 Schedule: Удалена запись из расписания доктора {appointment.doctor.name}")