

1. Opis projektu

Nazwa aplikacji: MedicalClinic - Aplikacja do zarządzania przychodnią lekarską

Cel aplikacji: Zarządzanie profilami pacjentów i personelu medycznego, w tym lekarzy oraz ich specjalizacji, w celu efektywnego świadczenia usług medycznych w placówce.

2. Wymagania funkcjonalne

Pacjenci:

1. Jako recepcjonista chcę mieć możliwość utworzenia profilu nowego pacjenta zawierającego jego dane osobowe oraz kontaktowe potrzebne do świadczenia usług medycznych.
 - a. **Metoda addPatient(Patient patient) w klasie PatientService:** Odpowiada za dodanie nowego pacjenta do systemu. Metoda korzysta z obiektu PatientRepository w celu zapisania informacji o pacjencie w bazie danych lub innym mechanizmie przechowywania danych.
2. Jako recepcjonista chcę mieć możliwość znalezienia pacjenta po numerze PESEL oraz wyświetlenia wszystkich jego danych.
 - a. **Metoda searchByPesel(String pesel) w klasie PatientService:** Odpowiada za wyszukanie pacjenta na podstawie numeru PESEL. Zwraca obiekt Optional<Patient>, co pozwala obsłużyć sytuację, gdy pacjent o podanym numerze PESEL nie istnieje w systemie. Wykorzystuje metodę findByPesel(String pesel) w klasie PatientRepository.
3. Jako recepcjonista chcę móc wyszukać wszystkich pasujących pacjentów o podanym nazwisku oraz wyświetlenia wszystkich danych znalezionych pacjentów.
 - a. **Metoda searchByLastName(String lastName) w klasie PatientService:** Odpowiada za wyszukiwanie pacjentów na podstawie podanego nazwiska. Zwraca listę obiektów typu Patient.

Personel medyczny:

1. Jako pracownik działu HR chcę mieć możliwość utworzenia profilu zatrudnionego lekarza uwzględniając wszystkie jego specjalizacje.
 - a. **Metoda addDoctor(Doctor doctor) w klasie DoctorService:** Odpowiada za dodanie nowego profilu lekarza do systemu z jego specjalizacjami.
2. Jako pracownik działu HR chcę mieć możliwość dodania nowej specjalizacji lekarzowi, który istnieje już w systemie.
 - a. **Metoda addSpecialtyToDoctor(UUID doctorId, MedicalSpecialty specialty) w klasie DoctorService:** Odpowiada za dodanie nowej specjalizacji do istniejącego profilu lekarza. Wyszukuje lekarza w systemie na podstawie jego id (UUID). Jeśli lekarz istnieje, specjalizacja jest dodawana do jego profilu, a zmodyfikowany obiekt lekarza zapisywany jest ponownie w DoctorRepository.
3. Jako recepcjonista chcę mieć możliwość znalezienia lekarza po jego numerze ID oraz wyświetlenia jego danych oraz specjalizacji.
 - a. **Metoda findDoctorById(UUID id) w klasie DoctorService:** Odpowiada za wyszukanie lekarza na podstawie jego id (UUID). Zwraca obiekt Optional<Doctor>

4. Jako recepcjonista chcę mieć możliwość znalezienia wszystkich lekarzy o określonej specjalizacji oraz wyświetlania ich danych oraz ID.

- a. **Metoda findDoctorsBySpecialty(MedicalSpecialty specialty) w klasie DoctorService:** Odpowiada za wyszukanie wszystkich lekarzy, którzy posiadają daną specjalizację. Zwraca listę obiektów Doctor, które spełniają kryterium specjalizacji.

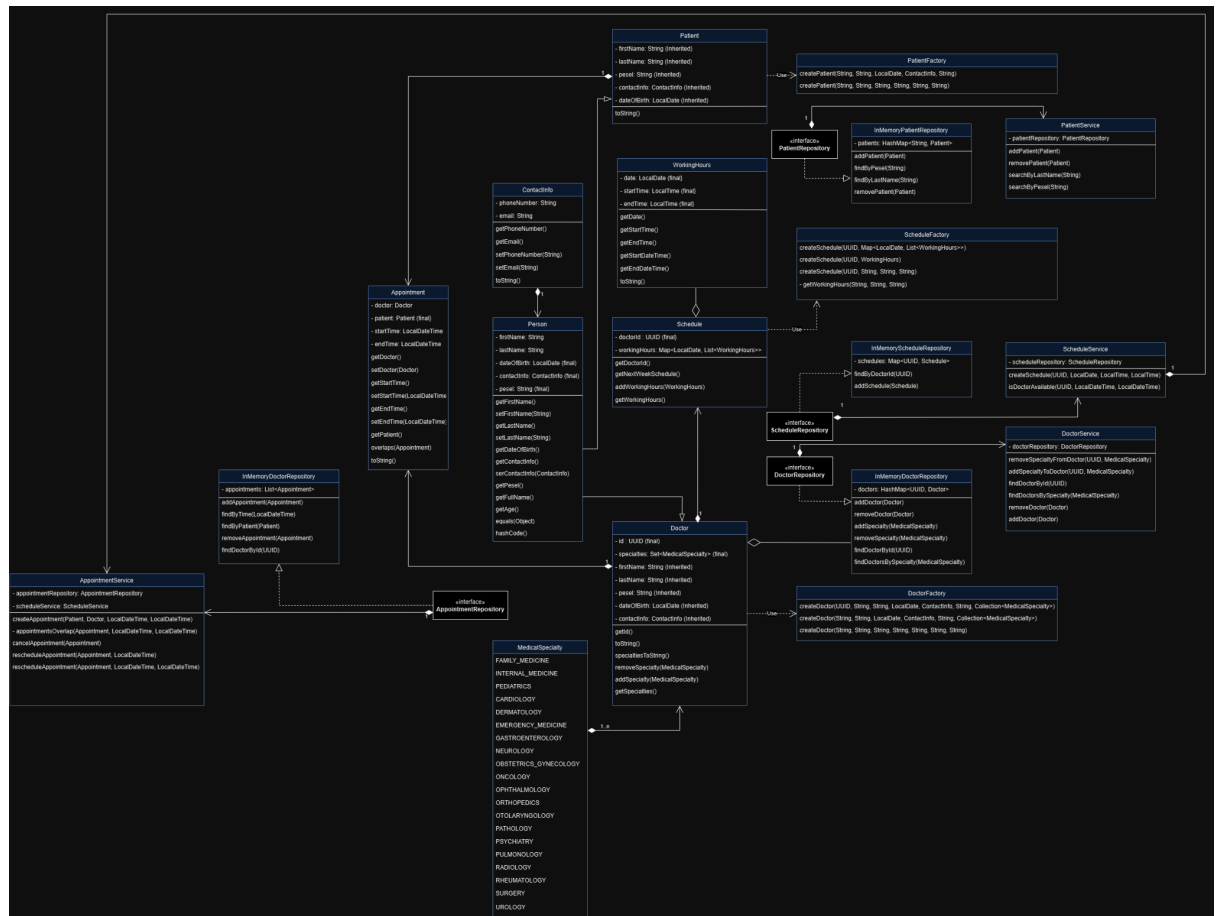
Grafik lekarzy:

1. Jako kierownik placówki chcę mieć możliwość tworzenia grafików dla wszystkich pracujących lekarzy. Pojedynczy grafik powinien zawierać godziny pracy wybranego lekarza w wybranym dniu.
 - a. **Metoda addWorkingHours(WorkingHours workingHours) w klasie Schedule:** Odpowiada za dodawanie godzin pracy dla lekarza w określonym dniu. Godziny pracy są walidowane za pomocą WorkingHoursValidator, aby zapewnić ich poprawność (np. brak nachodzenia na inne godziny pracy).
2. Jako pracownik recepcji chcę móc pobrać wszystkie utworzone grafiki wybranego lekarza na najbliższy tydzień, abym mógł sprawdzić w jakich godzinach przyjmuje pacjentów.
 - a. **Metoda getNextWeekSchedule() w klasie Schedule:** Odpowiada za pobieranie godzin pracy lekarza na najbliższe 7 dni. Rozpoczyna od aktualnej daty (today) do końca kolejnego tygodnia (endOfNextWeek). Zwraca listę wszystkich godzin pracy (List<WorkingHours>) dla lekarza w tym okresie.

Wizyty lekarskie:

1. Jako pracownik recepcji chcę mieć możliwość umówienia pacjenta w wybranym dniu o określonej porze na wizytę lekarską z wybranym lekarzem.
2. Wizyta lekarska powinna zostać pomyślnie umówiona tylko i wyłącznie w godzinach pracy wybranego lekarza (na podstawie jego grafiku). W przeciwnym wypadku powinien zostać wyrzucony wyjątek, a wizyta nie powinna zostać utworzona.
3. Ponadto wizyta powinna być pomyślnie umówiona tylko wtedy, gdy w tym samym czasie wybrany lekarz nie ma już innej wizyty. W przeciwnym wypadku powinien zostać wyrzucony wyjątek, a wizyta nie powinna zostać utworzona.
 - a. **Metoda createAppointment(Patient patient, Doctor doctor, LocalDateTime startTime, LocalDateTime endTime):** Sprawdza dostępność lekarza w podanym przedziale czasowym, korzystając z metody isDoctorAvailable() z klasy ScheduleService. Jeśli lekarz nie jest dostępny (poza godzinami pracy lub ma inną wizytę), wyrzuca wyjątek AppointmentException z komunikatem DOCTOR_NOT_AVAILABLE. W przypadku spełnienia warunków tworzy obiekt wizyty (Appointment) i dodaje go do repozytorium wizyt (AppointmentRepository).

3. Diagram klas [\(\[draw.io\]\(https://draw.io\)\)](https://draw.io)



4. Opis klas

Opis klasy **Person**:

Klasa **Person** reprezentuje osobę w systemie, mającą podstawowe dane identyfikacyjne i kontaktowe. Posiada atrybuty: `firstName` (imię), `lastName` (nazwisko), `dateOfBirth` (data urodzenia), `contactInfo` (dane kontaktowe) oraz `pesel` (numer PESEL). Konstruktor klasy przyjmuje te dane jako argumenty i inicjalizuje odpowiednie pola. **Person** zawiera metody do uzyskiwania i ustawiania tych atrybutów oraz do zwracania pełnej nazwy osoby oraz jej wieku. Metody `equals()` i `hashCode()` są odpowiedzialne za porównywanie i identyfikację osób w kolekcjach. Klasa posiada również metodę `toString()`, która zwraca sformatowaną łańcuchową reprezentację obiektu, zawierającą wszystkie istotne informacje.

Opis klasy **Patient**:

Klasa **Patient** dziedziczy po klasie **Person**, dodatkowo, **Patient** posiada własny konstruktor, który pozwala na utworzenie obiektu pacjenta, przekazując dane potrzebne do utworzenia osoby (`firstName`, `lastName`, `dateOfBirth`, `contactInfo`, `pesel`) oraz specjalny identyfikator pacjenta. Klasa ta posiada metodę `toString()`, która zwraca sformatowaną łańcuchową reprezentację obiektu **Patient**, zawierającą wszystkie istotne informacje o pacjencie.

Opis klasy **Doctor**:

Klasa **Doctor** dziedziczy po klasie **Person**, dodatkowo, **Doctor** posiada atrybut `id` (unikalny identyfikator lekarza) oraz `specialties` (zbiór specjalizacji medycznych, np. kardiolog, ortopeda). Konstruktor umożliwia utworzenie obiektu lekarza przez przekazanie odpowiednich danych osobowych oraz specjalizacji. Klasa ta posiada metodę `toString()`, która zwraca sformatowaną łańcuchową reprezentację obiektu **Doctor**, zawierającą dane osobowe oraz specjalizacje lekarza.

Opis klasy **Schedule**:

Klasa **Schedule** przechowuje harmonogram pracy lekarzy. Posiada atrybut `doctorId`, który identyfikuje lekarza, oraz `workingHours`, mapę przechowującą godziny pracy dla każdego dnia (`LocalDate`) oraz listę godzin pracy (`WorkingHours`). Konstruktor umożliwia utworzenie obiektu harmonogramu, przypisując do niego lekarza i godziny pracy. Klasa ta oferuje metody do zarządzania i wyświetlania godzin pracy, takie jak `addWorkingHours()`, która dodaje nowe godziny pracy, oraz

getNextWeekSchedule(), która zwraca harmonogram na nadchodzący tydzień. Metoda toString() zwraca sformatowaną łańcuchową reprezentację obiektu Schedule, zawierającą dane o lekarzu i godzinach pracy.

Opis klasy **WorkingHours**:

Klasa WorkingHours reprezentuje przedział godzin pracy dla konkretnej daty. Posiada atrybuty date (data), startTime (godzina rozpoczęcia) oraz endTime (godzina zakończenia). Konstruktor sprawdza, czy godzina rozpoczęcia nie jest późniejsza od godziny zakończenia, a następnie inicjalizuje obiekt. Klasa ta oferuje metody takie jak getStartDateTime() i getEndDateTime(), które zwracają pełne daty i godziny pracy. Metoda toString() zwraca sformatowaną łańcuchową reprezentację obiektu WorkingHours, zawierającą datę oraz godziny pracy.

Opis klasy **Appointment**:

Klasa Appointment reprezentuje wizytę pacjenta u lekarza w określonym czasie. Posiada atrybuty: doctor (lekarz), patient (pacjent), startTime (czas rozpoczęcia wizyty) oraz endTime (czas zakończenia wizyty). Konstruktor inicjalizuje te atrybuty przy tworzeniu nowego obiektu Appointment. Klasa ta oferuje metody do ustawiania i uzyskiwania tych wartości. Dodatkowo, posiada metodę overlaps() do sprawdzania kolizji terminów wizyt, oraz metodę toString(), która zwraca sformatowaną łańcuchową reprezentację obiektu Appointment, zawierającą dane o lekarzu, pacjencie oraz czasie wizyty.

5. Struktura pakietów

Projekt "MedicalClinic" jest podzielony na różne pakiety, które organizują funkcjonalności i komponenty aplikacji w sposób przejrzysty. Kluczowe pakiety to:

- **pl.wsb.lab.medicalclinic.patient:**
 - Zawiera klasy związane z pacjentami
 - Patient klasy odpowiadają za zarządzanie danymi osobowymi i kontaktowymi pacjentów.
- **pl.wsb.lab.medicalclinic.doctor:**
 - Zawiera klasy związane z lekarzami, takie jak Doctor oraz MedicalSpecialty.
 - Doctor zarządza danymi osobowymi lekarzy, identyfikatorem oraz specjalizacjami.
 - MedicalSpecialty to enum przechowujący różne specjalizacje medyczne, takie jak kardiolog, ortopeda, czy stomatolog.

- **pl.wsb.lab.medicalclinic.schedule:**
 - Zawiera klasy związane z harmonogramem pracy lekarzy, takie jak Schedule i WorkingHours.
 - Schedule przechowuje godziny pracy dla każdego dnia oraz pozwala na zarządzanie godzinami pracy lekarzy.
- **pl.wsb.lab.medicalclinic.appointment:**
 - Zawiera klasę Appointment, która zarządza wizytami pacjentów u lekarzy.
 - Appointment przechowuje informacje o lekarzu, pacjencie, czasie rozpoczęcia i zakończenia wizyty.
- **pl.wsb.lab.medicalclinic.shared.model:**
 - Zawiera wspólne klasy takie jak Person i ContactInfo, które są używane przez inne klasy w projekcie.