Covid Hospital Manager - Dokumentacja

Autorzy: Karol Orzechowski, Wiktor Pytlewski, Michał Kopeć

GitLab: https://gitlab-stud.elka.pw.edu.pl/korzech2/proi_21_covid_hospital_manager

Prowadzący: mgr inż. Justyna Stypułkowska

1. Założenia programu.

1.1. Podstawowe założenie programu

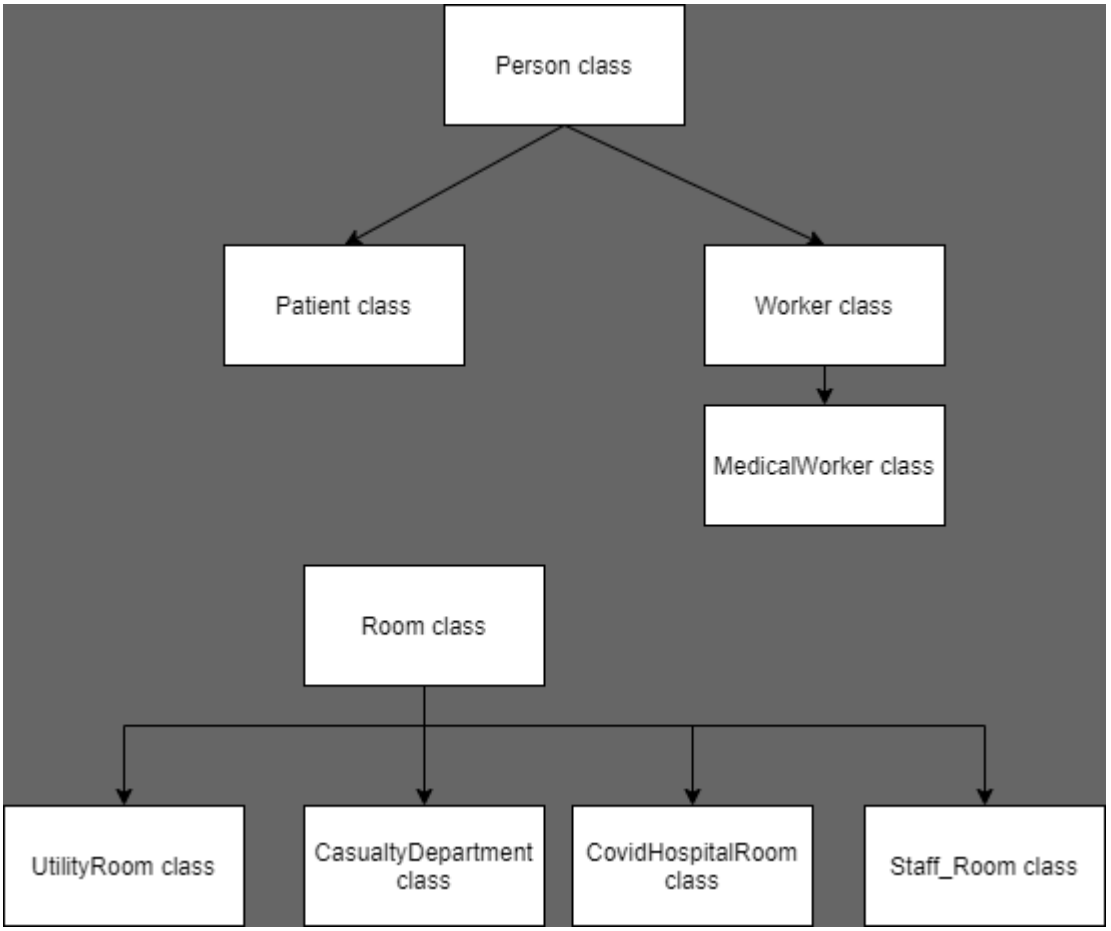
- Program symuluje pracę szpitala wyspecjalizowanego w leczeniu pacjentów chorych na COVID-19.

1.2. Pozostałe założenia

- Pracownicy szpitala pracują w dwóch możliwych **zmianach**:
  - poranna: 0:00-12:00
  - nocna: 12:00-24:00
- Każdy pacjent ma na **stałe przypisanego** indywidualnego lekarza i pielęgniarkę
- Szpital ma **jedną izbę przyjęć** i pacjenci mogą zostać dodani do szpitala tylko będąc wcześniej w izbie przyjęć.
- Pacjenci posiadają **jedynie 4 stany**:
  -dead (nieżywy)
  -critical (krytyczny)
  -stable (stabilny)
  -healthy(zdrowy)
- Lekarze muszą być osobami **powyżej 25 roku życia.**
- Nie ma możliwości włączenia programu podając jego parametry jako argumenty wywołania.
- Koniecznie jest **podanie ścieżki do pliku w formacie .json**, z którego czytane są parametry.
- W każdej wykonywanej iteracji do szpitala **przyjmowana jest stała ilość** pacjentów oraz wykonywane jest **jedno zdarzenie losowe**.
- Lekarzy nie może być mniej niż sal szpitalnych
- Lekarze nie mogą być zwalniani

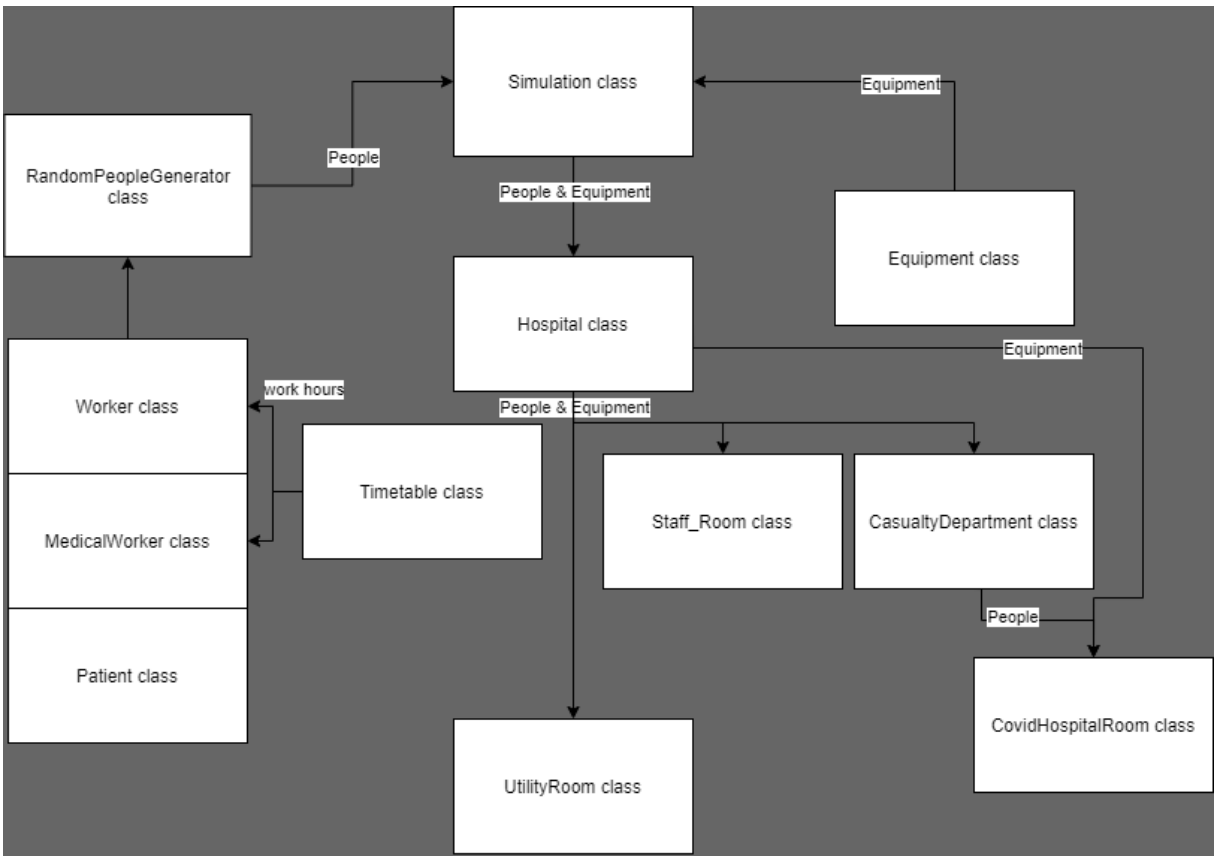
2. Architektura Rozwiązania

2.1 Schemat dziedziczenia w projekcie



2.2 Dekompozycja

- Uproszczony schemat przepływu informacji pomiędzy klasami.



2.3 Simulation Class

Jest to klasa odpowiedzialna za działanie symulacji szpitala. Tworzy ona obiekt klasy **Hospital**, odpowiada również za odczytanie parametrów startowych symulacji. Wykorzystuje ona **bibliotekę <random>**, w celu np. wylosowania jaka sytuacja ma się wykonać w danej iteracji. Jej destruktor odpowiada za usunięcie wszystkich losowo wygenerowanych osób.

2.4 Hospital Class

Klasa zarządzająca zasobami szpitala. Posiada ona zaimplementowany z biblioteki standardowej **kopiec maksymalny** sal szpitalnych (na szczycie kopca znajduje się zawsze sala szpitalna o największym współczynniku niezajętości łóżek). Każdy obiekt klasy Hospital posiada własny obiekt klasy **CasualtyDepartment,** który pełni rolę izby przyjęć. Dodatkowo klasa przechowuje w obiektach typu **std::queue<>** lekarzy oraz pielęgniarki (**MedicalWorker**).

Dzięki zapętleniu tych kolejek kadra medyczna jest przypisywana równomiernie do pacjentów (po przypisaniu do pacjenta członek personelu trafia na koniec kolejki).

Jednocześnie w odpowiednich obiektach klasy **std::vector<>** przechowywane są listy przyjętych pacjentów, odpowiednich sal oraz medyków.

2.5 Casualty Department Class

Klasa odpowiedzialna za przechowywanie pacjentów po ich przyjęciu (swoisty bufor na nich). W zależności od stanu przyjętego chorego, trafia on do odpowiedniej kolejki FIFO (**std::queue<>**).

2.6 Medical Worker Class

Odpowiada za implementację kadry medycznej. Dziedziczy ona po klasie **Worker.** Posiada ona obiekty klasy **std::vector<>** przechowujące wskaźniki na przypisanych pacjentów i przypisane sale. Dziedziczy po klasie **Worker**, która zawiera pole przechowujące obiekt klasy **Timetable,** Jest to klasa pomocnicza implementującą grafik pracownika medycznego.

2.7 Covid Hospital Room Class

Posiada obiekty **std::vector<>** dla pacjentów, lekarzy oraz pielęgniarek. Oblicza swój współczynnik niezajętości łóżek, jak inne klasy pochodne po klasie Room posiada **std::unordered_map<>** posiadającą jako klucz **enumerator** klasy Equipment, który jako wartość zwraca liczbę danych przedmiotów w pomieszczeniu.

2.8 Equipment Class

Klasa posiadająca enum opisujący różne obiekty. W celu umożliwienia wykorzystania jej w **std::unordered_map<>** posiada przeciążoną funkcję haszującą (liniową).

2.9 RandomPeopleGenerator Class

Klasa odpowiedzialna za generowanie losowych ludzi na podstawie pliku people.json. Plik bazowy generowany jest za pomocą krótkiego skryptu napisanego w Pythonie. Jako dane wejściowe bierze 3 pliki .txt z folderu source. Dane te następnie są losowo dobierane (z uwzględnieniem odpowiedniej płci pasującej do numeru pesel) i umieszczane w pliku json.

Klasa odczytuje plik json do obiektu **std::vector<>**, w którym znajdują się obiekty typu json, zawierające pesel, imię oraz nazwisko. Następnie te jsony zamieniane są w **std::unordered_map<>**, której kluczem jest **std::string** zawierający pesel, a wartością obiekt typu **std::pair<>** zawierający imię i nazwisko.

Podczas generowania ludzi, obiekty osób alokowane są dynamicznie, dodatkowo pracownicy muszą mieć więcej lat niż wartość pola *age_threshod.*

2.10 Pozostałe klasy

Pozostałe klasy zapewniają dodatkowe funkcjonalności dla wyżej wymienionych klas, jednakże nie są one na tyle interesujące aby o nich wspominać.

3. Aspekty Techniczne

3.1 Biblioteki oraz Inne Narzędzia

Podjęliśmy próbę zbudowania projektu za pomocą narzędzia typu make czy cmake, niestety bez większych sukcesów. Projekt wykorzystuje jedną zewnętrzną bibliotekę odpowiedzialną za obsługę plików typu json, jest to nlohmann/json (GitHub - nlohmann/json: JSON for Modern C++).

3.2 Instalacje

Dodana bibliotek nie wymaga instalacji, odpowiednie pliki nagłówkowe są już załączone w folderze include.

4. Testowanie programu

4.1 Symulacja

W celu symulowania działania szpitala tworzymy obiekt klasy **Simulation.** Zarządzając klasą **RandomPeopleGenerator generuje** ona obiekty potrzebne w symulacji szpitalu. Tworzy również obiekt klasy **Hospital** i korzystając z dostępnych funkcji symuluje jego pracę.

Symulacja opiera się na działaniu pętli **for**. Każda jej iteracja to imitacja wirtualnie mijającego czasu. W każdym kroku wykonywane jest **przyjęcie pacjentów z izby przyjęć** do szpitala oraz jedno z **losowych zdarzeń**:

- dodanie nowych pacjentów do izby przyjęć - **35%**
- polepszenie stanu zdrowia losowo wybranych pacjentów - **15%**
- wyzdrowienie losowo wybranych pacjentów - **15%**
- śmierć losowo wybranych pacjentów - **5%**
- pogorszenie stanu zdrowia losowo wybranych pacjentów - **10%**
- przyjęcie nowego lekarza do pracy - **8%**
- przyjęcie nowej pielęgniarki do pracy - **12%**

Każde z wyżej wymienionych zdarzeń ma odpowiednią procentową szansę na zaistnienie, wyliczoną korzystając z biblioteki **<random>.**

Po upłynięciu zadanej liczby iteracji wyświetlany jest końcowy stan szpitala.

4.2 Wyjątki

Niekrytyczne elementy programu zazwyczaj obsłużone są odpowiednią instrukcją warunkową. W sytuacji, gdy błąd jest krytyczny, zostaje rzucony jeden z własnych napisanych wyjątków np. jeżeli w toku działania programu liczba łóżek w sali szpitalnej będzie mniejsza od zera (`InvalidBedsNumberException`) albo w pokoju będzie więcej łóżek niż pacjentów (`RoomOverflowException`).

Dodatkową sytuacją wyjątkową jest **przepełnienie się szpitala.** Obsługa tego wyjątku polega na wypisaniu komunikatu o przepełnieniu szpitala, odrzuceniu dodania pacjenta i dalszym działaniu programu. Może się zdarzyć, że w następnej iteracji zwolnią się łóżka i pacjenci będą mogli być przyjmowani, więc nie ma potrzeby kończenia działania programu w tej sytuacji.

4.3 Testy Jednostkowe

Główną metodą testowania były testy jednostkowe, zawarte w folderze **Tests.**

4.4 Wnioski z Testowania

Uruchomienie graficznego debuggera współpracującego z visual studio code zdecydowanie ułatwiło poszukiwanie błędów w programie.

5. Instrukcja obsługi programu

5.1 Przykładowe Polecenie Startowe

Przykładowe polecenie do skompilowania programu dla systemu linux: `g++ --std=c++2a -Wall -Wextra `find -name '*.cpp'` -o main.out` (uruchamiamy z poziomu pliku main.cpp).

Następnie z poziomu pliku main.cpp uruchamiamy skompilowany plik (np. main.out).

5.2 Plik konfiguracyjny dla symulacji

W pliku parameters.json można ustawić wartości następujących parametrów:

- **number_of_rooms** - liczba sal szpitalnych
- **number_of_doctors** - początkowa liczba lekarzy (może wzrosnąć w trakcie symulacji)
- **number_of_nurses** - początkowa liczba pielęgniarek (może wzrosnąć w trakcie symulacji)
- **number_of_stable** - ile spośród sal szpitalnych ma być przeznaczona dla pacjentów w stanie stabilnym (pozostałe zostaną przeznaczone dla pacjentów w stanie krytycznym)
- **patients** - początkowa liczba pacjentów w szpitalu
- **number_of_actions** - ile dni ma trwać symulacja
- **number_of_patients_in_casualty_department** - maksymalna liczba pacjentów, którą może przyjąć izba przyjęć
- w części **equipments** można ustawić odpowiednią liczbę sprzętów dla poszczególnych sal (**mds** - mobilne stacje do dezynfekcji, **comp** - komputery, **dhs** - węzeł ciepłowniczy)

6. Wnioski z Projektu

Projekt niespodziewanie urósł do nieoczekiwanych rozmiarów (sumarycznie 3688 linijek kodu). Samo wymyślenie odpowiednich mechanizmów nie było problematyczne, jednakże niezliczone błędy w implementacji zdecydowanie wydłużyły proces produkcji oprogramowania. Podsumowując wiele spędzonych godzin z projektem wpłynęło bardzo owocnie na naszą ogólną wiedzę z programowania oraz samego języka C++.