SYSTEM INFORMACJI LEKARZA, PACJENTA
I APTEKARZA

"DKM-HEALTHCARE"

ANALIZA I DOKUMENTACJA PROJEKTU

Dorota Kapturkiewicz
Michał Świętek
Kamil Jarosz

1. MOTYWACJA

Nasz projekt, jak sama nazwa wskazuje, ma być systemem informacyjnym dla lekarzy, pacjentów i aptekarzy.

Skąd w ogóle taki pomysł?

Po pierwsze czegoś takiego bardzo brakuje. Z opieką medyczną związane jest mnóstwo dokumentów, jak chociażby cała historia choroby. Czemu zatem nie trzymać elektronicznie katalogu z historią dla każdego pacjenta? Poza tym, jakby lekarze mogli wystawiać recepty i skierowania drogą elektroniczną, byłoby to o wiele wygodniejsze. Nie trzeba by np. chodzić do przychodni z byle powodu, takiego jak potrzeba wystawienia recepty na lek, który i tak przyjmujemy regularnie i nie ma potrzeby, aby lekarz nas rzy tym badał. Ponadto, gdy do szpitala trafiają osoby z nagłymi przypadkami, lekarze nie będący lekarzami prowadzącymi dla danej osoby mogłyby sprawdzić wszystkie najważniejsze informacje, które mogą być kluczowe przy leczeniu (np. alergie, przyjmowane leki, itp.) oraz od razu dostać kontakt do lekarza rodzinnego. Poza tym czemu nie ułatwić ludziom przeglądania swojej historii choroby (to znaczy np. też sprawdzania wyników badań)? Przecież byoby wspaniale, gdyby każdy mógł ściągnąć sobie z bazy danych folder ze swoją historią. A jeszcze jakby do tego dostwał na maila powiadomienia o zmianach, to już w ogóle byłoby doskonale.
A co, jak idąc do apteki zapomnimy wziąć ze sobą receptę? Cała wycieczka na nic! Za to jakby recepta była w systemie, moglibysmy sobie w aptece sprawdzić jej numer, podać aptekarzowi, ten zaznaczyłby sobie jako zrealizowaną i wszystko poszłoby gładko.

W związku z tym postanowiliśmy zrobić małą próbkę takiego systemu.

UWAGA! Nie jesteśmy kilkudziesięcio osobowym zespołem, nie mamy roku i nikt nam za to nie płaci, w związku z tym nasz program jest tylko małą próbką tego, co powinno być zaimplementowane na potrzeby uzytkownika. W szczególności interfejs dla użytkownika jest konsolowy.

2. WSTĘPNA ANALIZA

2.1) Jak wyobrażamy sobie działanie aplikacji?

Każda osoba może zgłosić u lekarza rodzinnego, że chce dostać konto systemie, wtedy lekarz rodzinny kontaktuje się z administratorem i konto zostaje założone (hasło do odebrania u lekarza rodzinnego). Oczywiście później można to hasło zmienić.

Pacjent może po zalogowaniu do systemu dowiedzieć się różnych rzeczy dotyczących jego zdrowia (recepty, skierowania, alergie, prowadzący lekarze specjaliści itp). Może także ściągnąć swoją historię choroby jako folder zip. Tam też znajdują się wszystkie wyniki jego badań. Ponadto, mając skierowanie na jakis zabieg, może znaleźć numery telefonów do placówek w swoim mieście, które dane zabiegi wykonują.

Jeżeli osoba która się loguje jest dodatkowo aptekarzem, może (jak pozna nr recepty) "zrealizować" podana przez pacjenta receptę.

Osoba będąca lekarzem, może dodatkowo dowidziec się wszystkiego o swoich pacjentach, wystawiać recepty, skierowania, a także modyfikować historię

choroby i podstawowe ważne informacje. Poza tym ma wgląd i możliwość aktualizowania swoich miejsc pracy. W systemie zawarte są także informacje o jej/jego specjalnościach oraz prawach do wykonywania zawodu.

Ponadto system wysyła powiadomienia do odpowiednich osób jeśli ich historia choroby uległa zmianie w danym dniu lub wystawione i niezrealizowane recepty mają się przeterminować danego dnia lub za 5 dni (aby przypomnieć o konieczności ich realizacji). System także kontroluje serwery baz danych i w razie wystąienia problemów wysyła powiadomienia do administratorów.

2.2) Wymagania funkcyjne względem aplikacji

1. Autoryzacja
2. Przesyłanie danych
3. Prezentacja danych
4. Komunikacja z bazą danych
5. Kompatybilność dwóch (lub więcej) serwerów baz danych
6. Kontrola działania serwerów
7. Wysyłanie wiadomości (email)
8. Obsługa wielu klientów jednocześnie

2.3) Proponowane rozwiązanie - ideologia

Jedną z ważniejszych cech aplikacji "dkm-healthcare" jest **dostępność**. Użytkownicy powinni móc korzystać z systemu o każdej porze i każdego dnia. W związku z tym niedopuszczalne byłoby zrobienie tylko jednego serwera bazy danych. Nasze rozwiązanie proponuje stworzenie jednego, głównego serwera bazy danych, do którego kierowany byłyby zapytania, oraz drugiego, zapasowego, do którego kierowane byłyby komunikaty, jakby pojawiły się problemy na pierwszym. Oczywiście oba serwery powinny zachować **integralność**, to znaczy, że jeśli na jednym została dokonana modyfikacja bazy danych, drugi powinien to odnotować najszybciej jak to tylko możliwe. Nie rozwiązujemy tego poprzez sprawiedliwe rozdzielanie zapytań pomiędzy serwery, gdyż korzystamy z relacyjnej bazy danych, której nie trzymamy w sposób rozproszony (chociaż zdajemy sobie sprawę, że w "prawdziwym" projekcie zapewne należałoby tak właśnie zrobić, używając być może nierelacyjnej bazy danych).
Jednak, aby system był w pełni funkcjonalny, jeśli pojawi się awaria któregokolwiek z serwerów, powinniśmy poinformaować o tym osoby odpowiedzialne za utzrymywanie funkcjonalności systemu. To zadanie należy do serwera powiadomień, który to co godzinę sprawdza, czy serwery baz danych wciąż nasłuchują nowych zapytań i jeśli nie uda mu się nawiązać połączenia z którymkolwiek z serwerów baz danych, wysyła wiadomość do administratora. Jednak, aby już w pełni zapewnić dostęność, także serwer powiadomień ma swojego "stróża", który w razie niewydolności serwera powiadomień przejmuje jego funkcje.
Serwery powiadomień dodatkowo mogą wysyłać powiadomienia do użytkowników, o których pisaliśmy na pozcątku tego rozdziału.

W systemie informacji medycznej niezwykle istotna jest **poufność**. W końcu dane medyczne i historia choroby są ściśle tajnymi informacjami. Stąd też użytkownicy mają swoje hasła, a autoryzacja zapytań polega na zastowaniu **schematu Lamporta**. Ponadto komunikacje z serwerami baz danych odbywa się przez sockety SSL.

Uwaga! Jako że nie jesteśmy firmą klucze SSL mamy zahardcodowane w kodzie aplikacji.

2.4) Proponowane rozwiązanie - wykonanie

Aplikację "dkm-healthcare" postanowiliśmy napisać używając języka programowania Java, gdyż jest on doskonały do komunikacji sieciowej oraz obsługi baz danych. A także mamy z nim najwięcej doświadczenia.

Ponadto zostało założone specjalne konto na Gmail do obsługo aplikacji:
dkm.dkmhealthcare@gmail.com
Z tego konta wysyłane są wszystkie powiadomienia.

Strukturę aplikacji (serwery) przedstawiamy na rysunku *Rys1*.
Składa się ona z dwóch serwerów bazy danych (serwer główny (BD1), backup(BD2)) oraz dwóch serwerów do wysyłania powiadomień (N, NG).

Działają w niej następujące protokoły warstwy aplikacji:

P1. Komunikacja między klientem a serwerem (serwerami) bazy danych.
(programista - Kamil Jarosz)

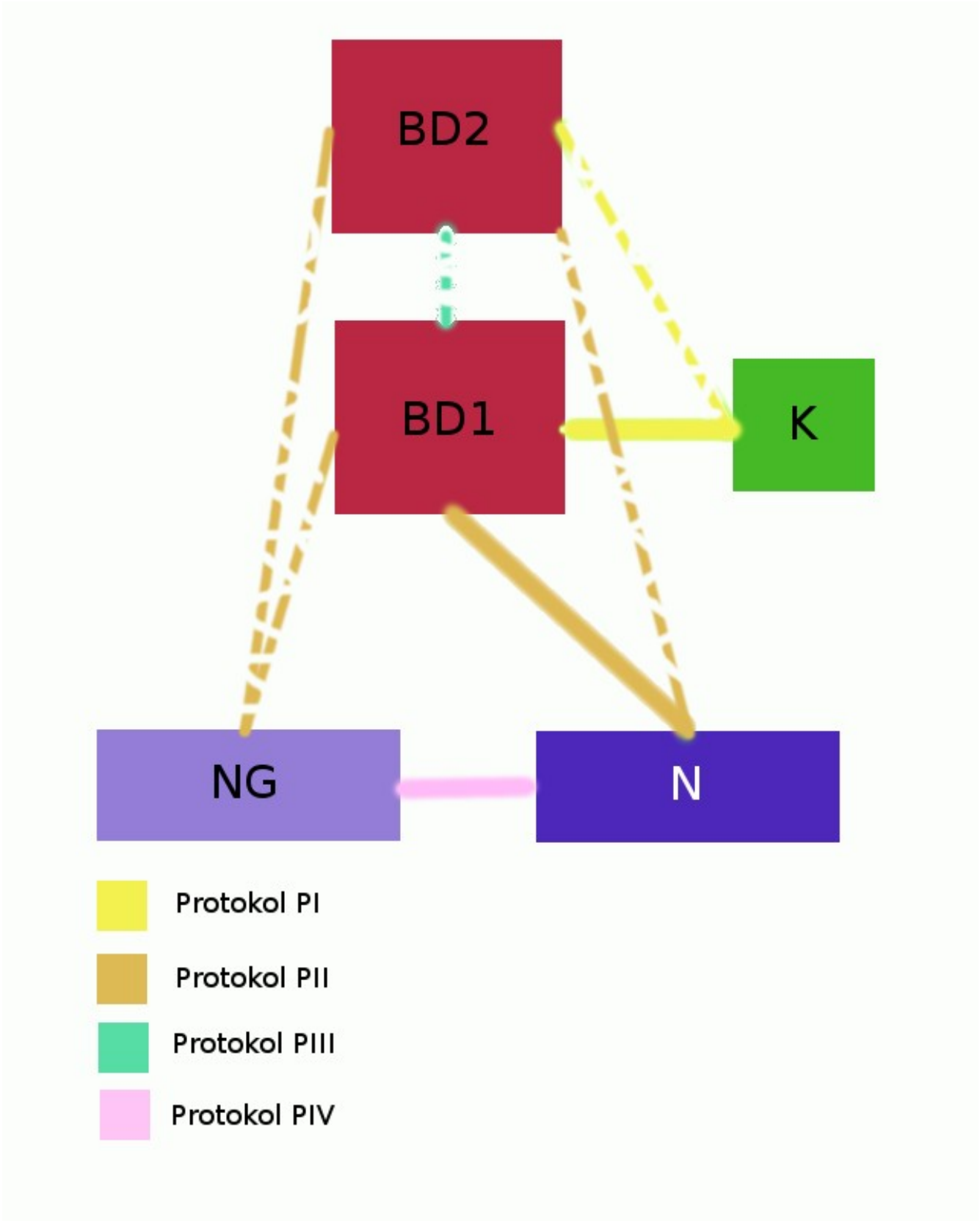
P2. Komunikacja między serwerami baz danych.
(programista - Kamil Jarosz)

P3. Komunikacja między serwerami baz danych a serwerami powiadomień.
(programista - Dorota Kapturkiewicz)

P4. Komunikacja między serwerami powiadomień.
(programista - Dorota Kapturkiewicz)

Przy okazji warto wspomnieć o pozostałym **podziale pracy**:

- Serwery baz danych - Kamil Jarosz
- Serwery powiadomień - Dorota Kapturkiewicz
- Interfejs użytkownika - obsługa lekarza i aptekarza oraz "strona startowa" - Dorota Kapturkiewicz
- Interfejs użytkownia - obsługa pacjenta - Michał Świętek
- Baza danych - Dorota Kapturkiewicz i Michał Świętek
- Dokumentacja - Dorota Kapturkiewicz


BD2
BD1
K
NG
N
Protokol PI
Protokol PII
Protokol PIII
Protokol PIV

Rys1.

2.5) Zarys działania protokołów:

P1.

"Logowanie" odbywa się przy pomocy schematu Lamporta. Następnie przy każdym zapytaniu przesyłamy (n+1) iterację hashu hasła użytkownia. Uzytkownik w protokole przesyła w większości zapytania sql do bazy. Jeśli były to zapytania typu "select", protokół otwiera nowe połaczenie i przesyła do uzytkownika obiekt będący odpowiednim opakowaniem rezultatu.
Ponadto uzytkownik może ściągnąć folder z historą choroby (jako archiwum zip), jesli zaś jest lekarzem może przesyłać na serwer pliki, któr chce dołączyć do historii choroby swojego pacjenta.

P2.

Gdy tylko nastąpi zmiana w bazie danych na serwerze BDi, wysyła on pinga" do serwera BD(i^1), a jak "ping" się powiedzie, to przesyła powiadomienia o zmianie. Serwer "otrzymujący" wiadomości dokonuje odpowiedniej aktualizaji po czym wysyła potwierdzenie serwerowi, od którego otrzymał wiadomość. Gdy serwer DB(i^1) nie odpowiada na pinga, serwer DBi pinguje aż się uda, w międzyczase zapisując zmiany do jakiegoś pliku tekstowego, który wyśle, gdy serwer DB(i^1) powróci. (Serwer wchodzący online czeka na plik tekstowy ze zmianami zanim zacznie obaługiwać klientów).

P3.

Serwer N (lub w przypadku sytuacji (*) NG) raz dziennie pyta serwer BD1 o wszytskich pacjentów, u których w okresie od ostatniego pytania wystąpiły zmiany w historii choroby, po czym wysyła powiadomienia mailem do tych pacjentów (protokół SMTP, biblioteka JavaMail). Ponadto raz dziennie pyta o recepty, które przeterminują sie dokładnie za 5 dni lub dzisiaj, i także wysyła odpowiednie powiadomienia. Jeżeli jest problem z połączeniem się z BD1, N próbuje zapytać o to

samo BD2.

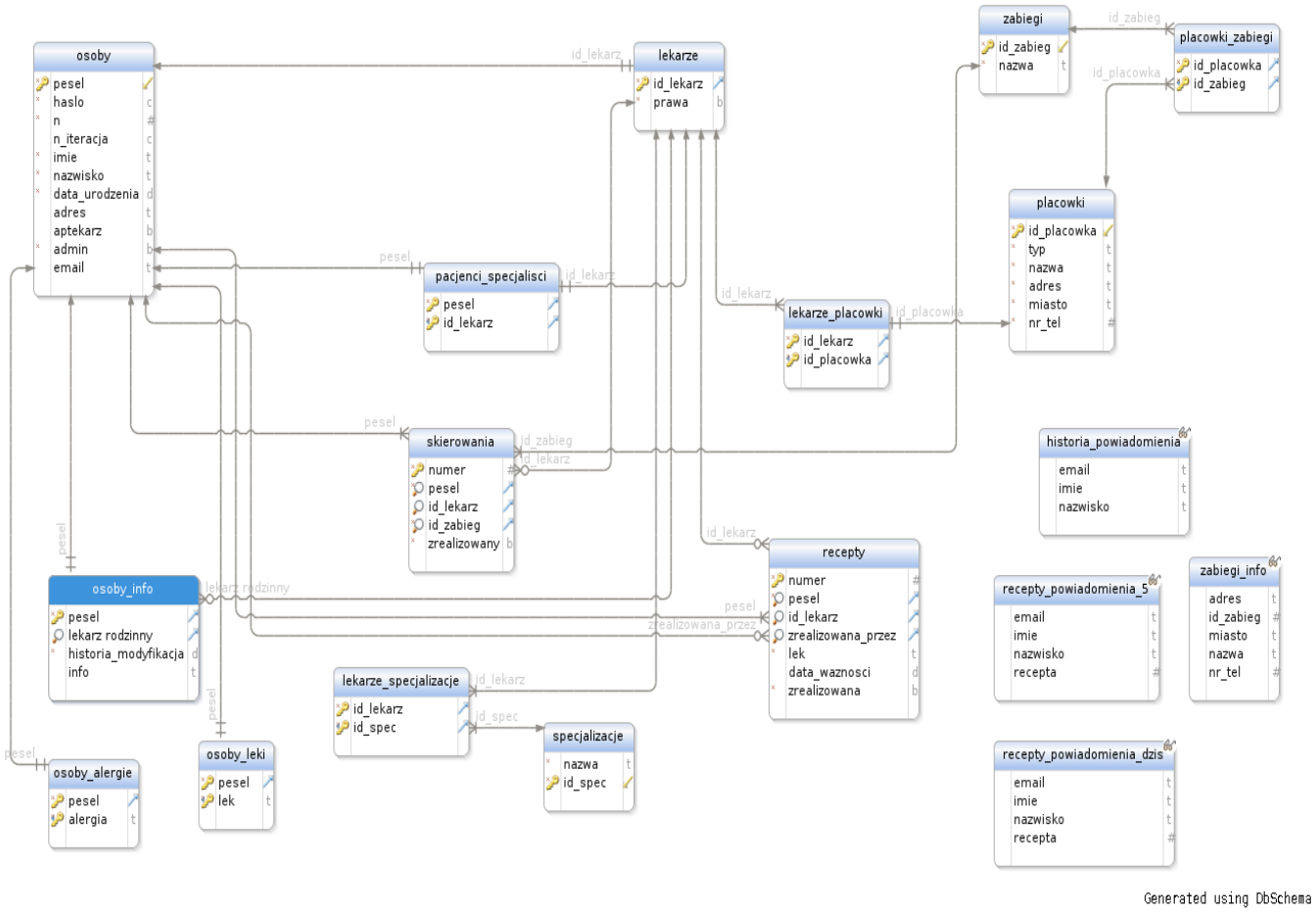
Ponadto, N co godzinę pinguje oba serwery baz danych (przez 100 razy co 3 sekundy lub do osiągnięcia sukcesu) i jeśli z któryms serwerem się nie udało, od razu wysyła widomość do administratorów.

P4.

Zawsze 5 min przed planowaną akcją serwera N, serwer NG wysyła "pinga" do N żeby sprawdzić, czy u niego wszystko w porządku. Jeśli się nie uda, przez następne 3 min NG próbuje co 5 sekund połaczyć się z N (jak się uda to przestajemy). Jeśli się nie udało, wtedy własnie mamy sytuację (*) i NG wykonuje zadanie N.

2.6) Baza danych

Schemat bazy danych przedstawiamy na rysunku *Rys2*.



Rys2.

3. ANALIZA I DOKUMENTACJA TECHNICZA

3.1) Protokół P3 i Główny Serwer Powiadomień (NServer)

Jak już wspomnieliśmy wcześniej, chcielibyśmy żeby nasza aplikacja miała funkcjonalność wysyłania do pacjentów powiadomień dotyczących wygasania terminów ważności niezrealizowanych recept oraz informowania, czy w danym dniu nastąpiła zmiana w historii choroby (bo to może świadczyć o np. pojawieniu się wyników badań).
Poza tym trzeba przypomnieć, że serwer powiadomień jest "strażnikiem" serwerów baz danych- co pewien czas sprawdza, czy serwery "żyją" - to znaczy, czy nasłuchują przychodzących połączeń. Jeśli nie, NServer wysyła wiadomość do administratora.

Do tego celu służy właśnie protokół P3. Ma on następujące cechy:

1. **ASYMETRIA**

Nasz protokół jest niesymatryczny. To znaczy, posługują się nim serwery powiadomień, które to zawsze rozpoczynają "rozmowę", oraz serwery baz danych, które są stroną odbierającą połączenie. Jest to zrobione w ten sposób, gdyż serwery baz danych są generalnie dużo bardziej obciążone niż serwery powiadomień, stąd nasza decyzja.
Dlatego też w klasie P3protocol zaimplementowany jest publiczny statyczny enumerator Site, i pole tego własnie typu, którego wartość nadaje się w trakcie tworzenia instancji klasy. Dzięki temu w trakcie "rozmowy" (funkcja talk) możemy odwoływać się od razu do informacji, którym serwerem obecnie jesteśmy (klasę P3protocol tworzy odpowiednio dla siebie serwer powiadomień

przed rozpoczęciem rozmowy i serwer baz danych po odczytaniu z socketa informacji, że interakcja ma być dokonywana wedle prorokołu P3).

2. **WARSTWA SIECI - TCP**

Protokół P3 oparty jest na protokole TCP w warstwie sieci.

3. **SOCKETY i STRUMIENIE DANYCH**

Rozmowa wedle protokołu P3 odbywa się poprzez odbieranie i wysyłanie strumienia danych do Socketa. Strumień ten "opakowany" jest odpowiadnio w klasy PrintWriter i BufferedReader, gdyż porozumiemy się znakowo, poprzez prowadzenie "rozmowy". W przypadku pytań o konkretne informacje z bazy danych, w pewnym momencie serwer Bazy Danych przesyła obiekt klasy CachedRowSetImpl (który jest serializowalnym rozszerzeniem klasy ResultSet, co umożliwia przesyłanie go poprzez ObjectInputStream i ObjectOutputStream). Do tego serwer powiadomień tworzy na chwilę ServerSocket, podaje jego adres w protokole tekstowym, a serwer baz danych otwiera nowe połączenie do tego socketu i przesyła obiekt (otwierane są strumienie do przesyłania obiektów). Jest to wzorowane na działaniu protokołu FTP, który to także otwiera osobne połączenie do przesyłania pliku. Aby rozróżniać rozmowy "pingujące" od próśb o konkretne informacje, stworzony jest statyczny enum Request, który przekazujemy do funkcji talk(). Ponadto, wszystkie Sockety których używamy, są Socketami SSL. Ale dlaczego, to o tym będzie dalej.

4. **STANOWOŚĆ**

Protokół P3 jest protokołem stanowym. Jest to wygodne, gdyż rozmowy są stosunkowo krótkie i nie kosztuje nas wiele pamiętanie w jakim momencie rozmowy jesteśmy. Bardziej obciążające byłby wysyłanie za każdym razem wiekszej ilości informacji. Ponadto Sockety idealnie się nadają do

"prowadzenia rozmów". Dlatego też powstała klasa ServerSocketThread, która każdą rozmowę otwiera w osobnym wątku, gdyż obsługa jednego klienta jest względnie długa (ale o niej później).

5. **POUFNOŚĆ**

Protokół P3 po stronie bazy danych sprawdza, czy pytającym jest serwer powiadomień. Serwerów powiadomień nie jest wiele, stąd też klasa protokołu zna adresy IP serwerów powiadomień (dostaje je w kondtruktorze wraz z obiektem klasy Config, który to przy tworzeniu czyta dane z pliku konfiguracyjnego. Po rozpoczęciu rozmowy (w trakcie "handshakingu") serwery baz danych sprawdzają czy adres IP pytającego zgadza się z którymś ze znanych mu adresów serwerów powiadomień. Jeśli tak nie jest, rozmowa zostaje przerwana. Ponadto poufność zapewnia także szyfrowanie połącznia, tzn. używanie klasy SSLSocket zamiast Socket.

6. **DOSTĘPNOŚĆ**

Nie chcielibyśmy, aby z jakiegoś błahego powodu serwer powiadomień niepokoił administartora informacjami o problemach z innymi serwerami lub z powodu chwilowej niedyspozycji któregoś z serwerów baz danych nie wysłał pacjentom obiecanych informacji. W związku z tym w klasie portokołu zaimplementowane są funkcje, których używają serwery powiadomień, aby w jak najbardziej wiarygodny sposób ściągać odpowiednie informacje z serwerów baz danych (funkcje pingServers i getInfo).
Generalnie podczas "pingowania" serwerów baz danych serwer powiadomień próbuje nawiązać połączenie i przeprowadzić udaną rozmowę wedle protokołu P3 z każdym serwerem 100 razy lub do skutku. Jeśli połączenie się nie uda, odczekuje on 3 sekundy przed ponowieniem próby (licząc na to, że serwer może ożyje). Dopiero po 100 nieudanych próbach serwer powiadomień "poddaje się" i powiadamia administratora, że coś jest nie w porządku.

Podobnie jest w przypadku ściągania odpowiednich informacji. Różnica jest jednak taka, że jeśli w przypadku "pingowania" chcemy zdobyć informację o każdym serwerze baz danych, tutaj wystarczy nam z jednego. Dlatego jeśli np uda nam się dostać to co chcemy z jednej bazy danych, pozostałych już nie niepokoimy. Dlatego też klasa P3protocol na początku swego istnienia tworzy z obiektu klasy Config odpowiednie listy adresów serwerów baz danych i serwerów powiadomień. Ponadto, przy otwieraniu drugiego połączenia do przesyłania obiektów, serwer baz danych próbuje wysłać obiekt klasy CachedRowSetImpl 10 razy lub do skutku.

7. **INTEGRALNOŚĆ**

Integralność jest zapewniona przede wszystkim poprzez sam fakt przesyłania obiektów. To znaczy, jeśli obiekt po przejściu rzez Socket wciąż się kompiluje, to mała szansa, że coś się w nim zepsuło. Ponadto Integralność zapewnia nam jeszcze używanie szyfrowenago połączenia (sockety SSL) - wtedy raczej nikt nam obiektu nie podmieni.

Dokładny przebieg "rozmowy" poprzez protokół P3 można wyczytać bezpośrednio z kodu źródłowego klasy.

Wszystkie powyższe funkcjonalności i cechy zaimplementowane sa bezpośrednio w klasie P3protocol. Dzięki temu serwer N nie musi się już troszczyć o implementację funkcji do poszczególnych WIELOKROTNYCH zapytań (to znaczy, to tego pytania do 100 razy o dane informacje, o których pisaliśmy w podpunkcie o dostępności). Robią to funkcje pingServers i getInfo w klasie P3protocol.

Chcielibyśmy natomiast, aby funkcje włączały się zawsze regularnie o odpowiednich godzinach. I oto dba już klasa NServer.
W związku z tym, pierwsze co trzeba było zrobić, to zadbać o dostęp do godzin, które nas inetersują (tj 1 w nocy oraz 4 po południu). Do tego służą obiekty klasy Calendar odpowiednio skonfigurowane i stworzone w instancji klasy Config. Dlatego też klasa NSerwer tworzy w konstruktorze taki obiekt. Do odpalania co pewien regularny odstęp czasu funkcji w nowym wątku świetnie nadaje się klasa Timer z biblioteki java.util. Jednak wymaga ona, aby ta odpalana funkca była funkcą run() z obiektu implemenującą klasę TimerTask. Stąd też NServer ma klasy wewntętrzne Notifier i Pinger, które rozszerzają TimerTask i uruchomiają odpowiednio metody getInfo i pingServers z klasy P3protocol.

Zadaniem serwera powiadomień jest jeszcze oczywiście wysyłanie wiadomości email. W tym celu korzystamy z biblioteki JavaMail, a klasa NSever posiada metody NotifyAdmin i NotifyPatients, które odpowiednio konstruują wiadomości. Sa one później wysyłane metodą SendMail, która już korzysta bezośrednio z metod i klas biblioteki JavaMail.

Serwer N musi także nasłuchiwać, gdyż serwer NG będzie z nim rozmawiał przy użyciu protokołu P4. Stąd też w klasie main znajduje się ServerSocket, który po nawiązaniu połączenia tworzy instancję klasy ServerSocketThread. Zauważmy, że komunikacja między serwerami powiadomień jest prosta i nie ma niej żadnych poufnych informacji, w związku z czym tu już nie używamy Socketów SSL, tylko zwykłe.

3.2) Protokół P4 i Strażnik Serwera Powiadomień (NGServer)

W rozdziale o protokole P3 i serwerze powiadomień opisaliśmy, jak to wszystko działa. Nie da się ukryć, że ktoś taki jak serwer powiadomień jest niezwykle potrzebny, gdyż to on pilnuje nam, aby ludzie (w szczególości administratozy) dowiadywali się o ważnych rzeczach. No ale co, jeśli coś się stanie naszemu serwerowi powiadomień? Wtedy nikt nas o tym nie poinformuje! Stąd też pomyśleliśmy o stworzeniu jeszcze jednego serwera, mianowicie Strażnika Serwera Powiadomień (NGServer). Jego zadaniem jest sprawdzanie zawsze na 5 minut przed planowaną akcją serwera powiadomień, czy wszystko z nim w porządku. Jeśli okaże się, że nie, wtedy NGServer wykonuje zadanie, które normalnie robi NServer (dopuszczamy możliwość, że działanie NServera zostanie zduplikowane- nie powinno dziać się to często, a lepiej dostać powiadomienie dwukrotnie aniżeli wcale).

Ponadto, raz na dobę NGServer wysyła administratorom wiadomość, że u niego wszystko dobrze.

Cała komunikacja pomiędzy głównym serwerem powiadomień a strażnikiem odbywa się przy pomocy prostego protokołu P4, który w zasadzie jest bardzo podobny do protokołu P3 (tylko o wiele krótszy). Warto jednak wspomnieć o jego następujących cechach:

1. **ASYMATRIA**
Podobnie jak P3, jest on niesymetryczny. To znaczy, rozpoczynającym rozmowę jest zawsze NGServer. Rozmowa polega na wymienieniu kilku zdań, przysłowiowym "spingowaniu" głównego serwera powiadomień.

2. **WARSTWA SIECI - TCP**

Protokół P4 oparty jest na protokole TCP w warstwie sieci.

3. **SOCKETY i STRUMIENIE DANYCH**

Rozmowa wedle protokołu P3 odbywa się poprzez odbieranie i wysyłanie strumienia danych do Socketa. Strumień ten "opakowany" jest odpowiadnio w klasy PrintWriter i BufferedReader, gdyż porozumiemy się znakowo, poprzez prowadzenie "rozmowy".

4. **STANOWOŚĆ**

Protokół P4 jest protokołem stanowym. Stosuje się tutaj taki sam argument jak w przypadku protokołu P3.

5. **POUFNOŚĆ**

W przypadku protokołu P4 nie potrzebujemy specjalnie martwić się o poufność. W końcu jedyna informacja jaką przez ten protokół dostajemy to to, czy nasz serwer żyje. Toteż nie mamy tutaj zaimplementowanych jakiś specjalnych zabezpieczeń.

6. **DOSTĘPNOŚĆ**

Jak już napisaliśmy wcześniej, dopuszczamy możliwość zduplikowania powiadomień. Chcielibyśmy jednak zredukować takie sytuacje do minimum. Stąd NGServer nie ogranicza się do jednej próby połączenia i przeprowadzenia rozmowy. W klasie P4protocol zaimplementowana została specjalna metoda (PingNServer), która próbuje połączyć się i porozmawiać z serwerem powiadomień 100 razy lub do skutku. Po każdym nieudanym połączeniu NGServer odczekuje 3 sekundy mając nadzieję, że NServer ożyje. Dopiero jak

100 razy poniesiemy porażkę, NGServer przejmuje działanie NServera oraz powiadomia administratora o niewydolności serwera powiadomień.

7. **INTEGRALNOŚĆ**

Ten protokół nie musi akurat specjalnie dbać o integralność. Jak już pisaliśmy wielokrotnie, informacje przesyłane przez niego nie są specjalnie ważne ani poufne.

Dokładny przebieg "rozmowy" poprzez protokół P4 można wyczytać bezpośrednio z kodu źródłowego klasy.

Klasa implementująca NGServer jest bardzo podobna do klasy NServer. Także posiada obiekt klasy Config oraz wewnętrzne klasy rozszerzające metody TimerTask - każda odpowiada innej działalności NServera, która powinna być zabezpieczona przez NGServer. Ponadto klasa NGServer posiada wewnątrz siebie obiekt klasy NServer, gdyż w razie niepowidzenia z "pingowaniem" serwera powiadomień odpala metody klasy NServer (które to nie są statyczne, stąd obiekt). Dodatkowo jeszcze w jedną z takich klas opakowane jest wysyłanie codzinnego raportu (tzn potwierdzenia, że się żyje) do administratora przez NGServer.
Z ciekawych rzeczy warto jeszcze wspomnieć, że NGServer wykorzystuje zdefiniowane w klasie Config obiekty klasy Calendar, aby rzeczywiście wykonywać "pingowanie" NServera na 5 minut przed zaplanowaną akcją.

3.3) Wątek serwera (ServerSocketThread)

Ze względu na **dostępność**, serwery muszą działać wielowątkowo. Nie ma możliwości, aby wszystko wykonywało się sekwencyjnie, gdyż wtedy nikt nie "dopchalby" się do bazy danych. Stąd postanowiliśmy zimplementować klasę ServerSocketThread, która dostaje w konstruktorze socket oraz kilka innych parametrów, w zależności od serwera, który tworzy instancję klasy (serwer bazy danych np. podaje także jako parametr obiekt klasy Connection, czyli połączenie z bazą danych - jest to potrzebne np. w protokole P3). Oczywiście klasa ServerSocketThread implementuje interfejs Runnable, aby mogła być odpalana jako osobny wątek.

Jest to pomyślane tak, aby serwer nasłuchiwał poprzez ServerSocket na jakimś porcie, i w momenci gdy ktoś sie z nim połączy i (dzięki metodzie accept) powstanie Socket do rozmowy, zostanie on bezpośrednio przekazany do nowej instacji klasy ServerSocketThread, natomiast pętla nasłuchująca będzie mogła działać dalej osobno.

W metodzie run Socket czyta pierwszą wiadomość, która w naszej aplikacji zawsze jest jedną z nazw protokołów (P1, P2, P3 lub P4), i w zależności od protokołu tworzy nową instancję klasy protokołu i w niej odpala metodę do dalszej komunikacji.

Uwaga: Wszędzie tutaj piszę Socket, ale klasie ServerSocketThread można równie dobrze dać SSLSocket, jako że SSLSocket dziedziczy po klasie Socket.

3.4) Interfejs Użytkownika (Client, Patient, Doctor, Pharmacist)

Interfejs użytkownika zaimplementowany jest w czterech klasach - Client, Patient, Doctor i Pharmacist. Klasy Patient, Doctor i Pharmacist są klasami obsługującymi odpowiednio pracę w trybie pacjenta, lekarza i aptekarza. Ich instancje tworzone są w metodzie main klasy Client (która to włącza się wraz z uruchomieniem programu), jeśli po zalogowaniu się użytkownik wybierze pracę w odpowiednim trybie (może to zrobić tylko wtedy jeśli ma takie uprawnienia - aplikacja dowiaduje się tego przy pomocy protokołu P1).

Każda z tych trzech klas (bez klasy Client) ma metodę perform, która to jest "pętlą while" pracy w danym trybie. Użytkownik zgodnia z wyświetlaną instrukcją powinien wpisywać odpowiednie liczby i w ten sposób sygnalizuje aplikacji, co by chciał i polecenia się wykonują.

3.5) Protokół P1 i komunikacja Serwera Bazy Danych (DBServer) z Użytkownikiem (Client)

Protokół P1 stanowi główną część aplikacji. To przez niego porozumiewają się użytkownik z serwerem bazy danych.
Protokół pozwala na pięć akcji - logowanie, wykonanie zapytania w bazie danych i zwrócenie wyniku, ściągnięcie własnej historii choroby oraz wysłanie na serwer pliku.
Żeby logowanie było bezpieczne, wykorzystujemy **schemat Lamporta**: baza danych przechowuje dla każdego klienta numer iteracji hasła i odpowiednie złożenie funckji md5 na haśle. W chwili, gdy użytkownik chce się zalogować,

baza danych wysyła mu numer iteracji - n. Użytkownik liczy u siebie złożenie (n-1) funckji md5 na haśle i odsyła serwerowi. Serwer bazy danych obkłada to jeszcze raz funcją md5 i porównuje z przechowywaną w bazie iteracją hasła. Jeżeli się zgadzają, serwer bazy danych updatuje w bazie n na wartość o jeden mniejszą i poprzednią iteracją hasła.

Protokół P1 ma nastepujące własności:

1. **ASYMETRIA**

Jedynym słusznym rozwiązaniem jest przyjęcie, że rozmowę zawsze rozpoczyna użytkownik. Dlatego protokół P1 ma własny enumerator Site, który mówi, czy jesteśmy uzytkownikiem, czy serwerem bazy danych. Funkcja talk służy do komunikacji i obsługuje zapytanie do bazy danych, wysyłanie i pobieranie plików. Do rozróżniania zapytań służy enumerator Request.

2. **TCP**

Protokół P1 oparty jest na protokole TCP w warstwie sieci.

3. **SOCKETY I STRUMIENIE DANYCH**

Komunikacja w protokole P1 opiera się na przesyłaniu strumieni danych do Socketa.

4. **STANOWOŚĆ**

Protokół P1 jest protokołem stanowym. Jest to wygodne, gdyż rozmowy są stosunkowo krótkie i nie kosztuje nas wiele pamiętanie w jakim momencie rozmowy jesteśmy. Bardziej obciążające byłby wysyłanie za każdym razem wiekszej ilości informacji. Dlatego też powstała klasa ServerSocketThread, która każdą rozmowę otwiera w osobnym wątku. Jest to też o tyle wygodne,

gdyż nie chcemy podawać informacji z bazy danych byle komu. Ale o tym w następnym podpunkcie.

5. **POUFNOŚĆ**

Protokół P1 po stronie użytkownika sprawdza, czy odpowiada mu serwer bazy danych. Serwerów powiadomień nie jest wiele, stąd też w protokole są zahardkodowane adresy IP baz danych. Ponadto przy logowaniu stosujemy opisany wcześniej schemat Lamporta.

6. **DOSTĘPNOŚĆ i INTEGRALNOŚĆ**

Może się zdarzyć, że serwer bazy danych ulegnie awarii. Nie chcemy, żeby wtedy użytkownicy stracili możliwość sprawdzenia np. kiedy mają zabieg! Właśnie dlatego mamy dwa serwery bazy danych, z którymi użytkownik może sie połączyć. Synchronizacją pomiędzy bazami danych zajmuje się protokół P2. Do przesyłania danych pomiędzy bazą danych a użytkownikiem uzywamy bezpiecznych SSLSocketów.

7. **SKALOWALNOŚĆ**

Dla wygody adresy serwerów baz danych są także zahardcodowane w klasie protokołu P1.
Na potrzeby skalowalności są jednak w liście i można łatwo podmienić, aby lista ta była wypełniana z jakiegoś pliku konfiguracyjnego.

4.6) Protokół P2 i synchronizacja baz danych

Niektórzy użytkownicy (Doktor, Farmaceuta) mają możliwość wprowadzania zmian w bazie danych. Ponieważ mamy dwa serwery baz danych, a

użytkownik łączy się tylko z jednym z nich, zmiany są widoczne tylko na tym serwerze, z którym połączył się użytkownik. Serwer baz danych, która została zmodyfikowana, próbuje wysłać tę samą modyfikację drugiemu serwerowi. Ale...

Czasami zdarza się awaria serwera baz danych. Ponieważ mamy dwa serwery, nie stanowi to zbytniego problemu: użytkownik może łączyć się z tym drugim serwerem w celu otrzymania informacji. Ale co jeśli użytkownik chce wprowadzać zmiany w bazie danych? Gdy awaria pierwszego serwera zostanie usunięta, będzie dokładnie w tym samym stanie, w jakim był przed awarią. Ale przecież użytkownik modyfikował zawartość bazy danych! W tej sytuacji trzeba synchronizować zawartość serwerów. Gdy serwer baz danych nie może wysłać modyfikacji do tego drugiego serwera, zapisuje sobie zapytania SQLa w pliku tekstowym (./log). Gdy serwer powróci po awarii, dostanie od sprawnego serwera tenże plik.

Protokół ten ma nastepujące własności:

1. **ASYMETRIA**
Zarówno pierwszy, jak i drugi serwer może ulec awarii. W funkcji "talk" protokołu udział biorą dwie strony: serwer, który rozpoczyna synchronizację (DBInit) oraz "ten drugi" (DBRec). Reprezentuje je enumerator Site.

2. **TCP**
Protokół P2 oparty jest na protokole TCP w warstwie sieci.

3. **SOCKETY I STRUMIENIE DANYCH**
Komunikacja w protokole P2 opiera się na przesyłaniu strumieni danych do

Socketa.

4. **STANOWOŚĆ**

Protokół P2 jest protokołem stanowym. Jest to wygodne, gdyż rozmowy są stosunkowo krótkie i nie kosztuje nas wiele pamiętanie w jakim momencie rozmowy jesteśmy. Bardziej obciążające byłby wysyłanie za każdym razem wiekszej ilości informacji. Dlatego też powstała klasa ServerSocketThread, która każdą rozmowę otwiera w osobnym wątku. Jest to też o tyle wygodne, gdyż nie chcemy podawać informacji z bazy danych byle komu. Ale o tym w następnym podpunkcie.

5. **POUFNOŚĆ**

Stosujemy bezpieczne gniazka SSL.

6. **DOSTĘPNOŚĆ i INTEGRALNOŚĆ**

Zapewnia je właśnie protokół P2.

7. **SKALOWALNOŚĆ**

Dla wygody adresy serwerów baz danych są także zahardcodowane w klasie protokołu P2.
Na potrzeby skalowalności są jednak w liście i można łatwo podmienić, aby lista ta była wypełniana z jakiegoś pliku konfiguracyjnego.

4. DOKUMENTACJA UŻYTKOWA

4.1) Uruchomienie aplikacji po stronie klienta

Aby uruchomić aplikacje po stronie klienta należy uruchomić metodę main z klasy Client. Program wtedy poprosi o podanie numeru pesel użytkownika oraz hasła. Aby otrzymać hasło do systemu (teoretycznie każdy obywatel państwa powinien mieć konto w systemie), należy zgłosić to swojemu lekarzowi rodzinnemu. Lekarz rodzinny powinien mieć w zamknietych kopertach wstępne hasła do kont swoich pacjentów.

Następnie (zgodnie ze swoimi rolami w systemie) można wybrać jeden z trybów pracy - tryb pacjenta, tryb lekarza oraz tryb aptekarza. Tryb pacjenta dostępny jest dla każdego użytkownika, zaś tryb lekarza tylko dla lekarzy (z aktualnym prawem wykonywania zawodu), natomiast tryb aptekarza tylko dla aptekarzy.

Można także zmienić hasło. Do tego trzeba wybrać odpowiedni (podany w menu) numer, po czym postępować zgodnie z wyświetlaną instrukcją.

4.2) Instrukcja obsługi programu dla pacjenta

W menu pacjenta znajdują się następujące możliwości:
- ściągnięcie historii choroby,
- wyświetlenie podstawowych informacji o pacjencie,
- wyświetlenie recept wypisanych na pacjenta,
- wyświetlenie skierowań posiadanych przez pacjenta,

- wyszukanie informacji o placówkach w zadanym mieście, w których wykonywany jest zadany zabieg,
- wyszukanie informacji o placówkach w zadanym mieście, w których przyjmują lekarze o zadanej specjalności,
- wyświetlenie informacji o lekarzu rodzinnym, do którego przypisany jest pacjent.

Aby ściągnąć historię choroby należy podać bezwzględną ścieżkę, gdzie ma zostać pobrana historia choroby.
Na wyświetlane informacje o pacjencie składają się: imię, nazwisko, pesel, data urodzenia oraz adres. Pacjent nie ma możliwości zmiany tych informacji.
Na wyświetlane informacje o receptach pacjenta składają się: nazwa I dawkowanie leku, data ważności oraz informacja czy dana recepta została zrealizowana. Informacje o receptach są posortowane w taki sposób, że najpierw wyświetlane są informacje o receptach niezrealizowanych, a następnie o receptach zrealizowanych.
Na wyświetlane informacje o posiadanych przez pacjenta skierowaniach składają się: numer skierowania, nazwa zabiegu na jaki dane skierowanie jest przepisane, data ważności skierowania oraz informacja czy dane skierowanie zostało zrealizowane. Informacje o skierowaniach są posortowane w taki sposób, że najpierw wyświetlane są informacje o skierowaniach niezrealizowanych, a następnie o skierowaniach zrealizowanych.
Aby znaleźć informacje o interesujących nas zabiegach w pierwszej kolejności wpisujemy nazwę miasta, a następnie wpisujemy nazwę interesującego nas zabiegu. Na każdym etapie mamy możliwość powrócić do menu głównego lub opuścić aplikację. Jeśli pacjent nie zdecyduje się opuścić aplikacji, to jego oczom ukaże się tabelka, w której znajdują się następujące informacje o placówkach wykonujących zadany zabieg w zadanym mieście: nazwa placówki, typ, adres oraz jakże istotny numer telefonu.

Aby znaleźć informacje o interesujących nas specjalistach zaczynamy od wpisania nazwy miasta, a następnie wpisania nazwy interesującej nas specjalizacji. Jak poprzednio, na każdym etapie mamy możliwość powrócić do menu głównego lub opuścić aplikację. Tabelka wyrysowująca się na ekranie zawiera w sobie następujące informacje o placówkach zatrudniających lekarzych parających się daną specjalizacją w zadanym mieście: nazwa placówki, typ, adres oraz jakże istotny numer telefonu.
Na wyświetlane informacje o lekarzu rodzinnym, do którego przypisany jest dany pacjent składają się: imię, nazwisko, nazwa placówki, w której pracuje lekarz rodzinny, do którego jest przypisany zalogowany aktualnie w aplikacji pacjent, adres tejże placówki oraz numer telefonu placówki, która zatrudnia powyższego lekarza rodzinnego.

4.3) Instrukcja obsługi programu dla lekarza

Po włączeniu trybu lekarza pojawia się menu trybu lekarza. Można tam wybrac różne opcje, zarówno odczytu jak i modyfikacji danych.
Lekarz ma możliwości:
- pracować z jednym konkretnym pacjentem
- wyświtlać dane pacjentów, dla których pełni funkcję lekarza rodzinnego lub specjalisty
- sprawdzić swoje prawa wykonywania zawodu
- wyświetlić i modyfikować swoje miejsca pracy

Lekarz także może dostać adres email administratorów, do których powinien wysłać wiadomość, jeśli chciałby zgłosić coś administratorom. Przykładem czegoś takiego jest sytuacja, gdy lekarz odnowił prawo wykonywania zawodu,

jednak z jakiś powodów nie zostało to odnotowane w bazie danych. Nie chcielibyśmy jednak, aby lekarz mógł sam sobie modyfikować uprawnienia - mogłoby to prowadzić to wielu oszustw!

Aby pracować z jednym pacjentem, należy wybrać odpowiednią opcję z menu trybu lekarza i (postępując zgodnie z instrukcją) podac pesel pacjenta.
Jeśli pacjent jest w jakiś sposób "zwązany" z danym lekarzem (to znaczy lekarz jest dla niego lekarzem rodzinnym lub zarejestrowanym specjalistą), lekarz ma dostęp do historii choroby.

Historia choroby trzymana jest na dysku serwera bazy danych w postaci folderu. Są w nim pliki tekstowe, zdjęcia RTG, wyniki badań i ogólnie cala historia. Aby ściągnąć historię choroby, lekarz powinien (znów wcześniej postępując zgodnie z instrukcją) podać ścieżkę dostępu do folderu, do którego chcielibyśmy ściągnąć historię (w postaci folderu zip). Aby załadować jakiś plik do historii, lekarz powinien także wybrać odpowiednią opcję z menu, po czym podać ścieżkę dostępu do pliku, który chciałby załadować.
Uwaga! Podane ścieżki muszą być ściezkami bezwzględnymi.

Zauważmy jednak, że lekarz może obsługiwać pacjenta losowego. Takie sytuacje mają miejsce na codzień np. na izbie przyjęć. Dlatego też dostęp do wglądu i modyfikacji takich danych pacjenta jak ważne informacje zdrowotne (tam np. zanotowane są takie rzeczy jak to, że pacjent jest po przeszczepie serca), alergii lub przyjmowanych lekach ma każdy lekarz. Lekarz nie będący prowadzącym może także dostać adres email do lekarza rodzinnego obslugiwanego pacjenta, aby ponformować go o jakiś ważnych wydarzeniach w życiu zdrowotnym pacjenta.

Alergii i przyjmowanych leków pacjenta nie można usuwać, jako że nie

chcemy, aby informacje o takich rzeczach przepadły. Aby dodać leki lub alergie, należy postępować zgodnie z wyświetlaną instrukcją.
Uwaga! Nie należy podawać pustych linii przed poadaniem nowego leku lub alergii! Lek trzeba podawać z dawkowaniem.

Ważne informacje o pacjencie mogą byc modyfikowane na dwa sposoby:
1. Poprzez dopisanie czegos na koniec informacji
2. Poprzez zmianę całej informacji
Odpowiednią opcję można wybrać w menu wyświtlania i modyfikacji informacji o pacjencie.

Pacjentowi można także wypisać receptę. Aby to uczynić, należy postępować zgodnie z instrukcją. To znaczy, należy wybrać odpowiednią opcję, po czym w jednej lini wrowadzić nazwę leku idawkowanie. Numer recepty oraz dane pacjenta zostaną wypełnione automatycznie.

Lekarz ma także możliwość wystawiania skierowania. Do tego także powinien postępować zgodnie z instrukcją. Jest to proces bardzo podobny do wystawiania recept. Róznica jest taka, że należy podać numer identyfikacyjny zabiegu, na jaki chcemy wystawić skierowanie. Przed wpisaniem tego można jadnak poprosić system (znowu opcja w menu), aby wyświetlił wszystkie dostępne zabiegi wraz z ich numerami, aby miało się z czego wybierać. Dodawać zabiegi do systemu może tylko administrator, toteż jeśli nie ma interesującego nas zabiegu, powinniśmy wysłać wiadomość do administartora.

Skierowanie można także realizować. Polega to na tym, że lekarz powinien dostać od pacjenta numer skierowania i postępując zgodnie z poleceniami systemu, wpisać ją. Jeśli skierowanie nie zostało już wcześniej zrealizowane,

operacje się powiedzie. Jeśli jednak skierowanie już ktoś wykorzystał, system powienien o tym poinformaować Dlatego, ważne jest, aby realizować skierowanie w systemie przed realizacją go w rzeczywistości! (Aby uniknąć nieuczciwości ze strony pacjenta).

Jak już zostało wcześniej wspomniane, lekarz ma możliwość wglądu i modyfikacji swoich miejsc pracy. Do tego także powinien postępowac zgodnie z instrukcją. Jeśli chce usunąc lub dodać miejsce pracy, powinien wprowadzić numer identyfikacyjny placówki. Jest to pierwsza wartość podczas wypisywania, także do usunięcia wystarczy najpierw wyświetlić placówki i sprawdzić numer. Jenak aby dodać nowe miejsce pracy, należy ten numer po prostu znać. Ale nie jest on żadną tajemnicą, także każda placówka powinna przekazywać nowozatrudnionemu swój numer.

Po skończeniu pracy w trybie lekarza, program wraca do menu głownego.

4.4) Instrukcja obsługi programu dla aptekarza

Po włączeniu trybu aptekarza pojawia się menu trybu aptekarza. Można tam wybrac opcję realizacji recepty lub powrót do menu głównego aplikacji.

Aptekarz ma możliwość "realizacji" recept. W tym celu klient musi podac mu numer recepty (nie chcielibyśmy, żeby aptekarze mieli możliwości wglądu ww wszystkie recepty pacjentów). Aby dokonać realizacji recepty (postępując zgodnie z wyświetlaną na ekran instrukcją) należy podać numer recepty pacjenta. Jeśli nie została zrealizowana wcześniej, recepta zostanie oznaczona

jako zrealizowana i automatycznie pesel danego aptekarza zostanie wpisany w kolumnę zrealizowana_przez w tabeli recept w bazie danych.
Po zrealizowaniu recepty program wraca do menu trybu aptekarza.

4.5) Uruchomianie serwerów i tworzenie pliku konfiguracyjnego

Na początku należy sprawdzić dostępność portów, na jakich chcielibyśmy uruchomić serwery. Następnie należy wypełnić plik konfiguracyjny (zawarty w źródłach projektu) w następujący sposób:

NServer [adres IP maszyny] [nr portu]
NGServer [adres IP maszyny] [nr portu]
BD1 [adres IP maszyny] [nr portu]
BD2 [adres IP maszyny] [nr portu]
password [hasło do konta administratora]

Następnie należy uruchomić serwery baz danych. Jeśli uruchomiamy je po raz pierwszy, nie podajemy argumentów dla klasy main. Jeśli jednak uruchomiamy serwer po awarii, należy podać argument "recovery", gdyż wtedy, zgodnie ze specyfikacją, serwer czeka na plik ze zmianami od drugiego serwera baz danych.

Dopiero po uruchomieniu serwerów baz danych należy uruchomić serwer powiadomień (NServer), a w ostatniej kolejności strażnika serwera powiadomień (NGServer).

UWAGA! Projekt używa Mavena do ściągania bibliotek I rozwiązywania zależności. Dlatego radzimy wszystko uruchamiać z poziomy edytora Eclipse ze ściągniętą wtyczką do Mavena. Wtedy wszystko ładnie działa.

4.6) Tworzenie bazy danych

Do projektu dołączamy plik Create.sql, który zawiera w sobie stworzenie bazy danych oraz wypełnienie przykładowymi danymi. Korzystamy z bazy Postgresql, zatem należy mieć ją zainstalowaną oraz mieć ściągnięte drivery jdbc. Ponadto należy zmodyfikować pliki źródłowe serwera bazy danych, aby łączył się z dobrym userem, dobra bazą danych I miał dobre hasło.

5. DOKUMENTACJA IMPLEMENTACYJNA

W ramach dokumentacji implementacyjnej załączamy wygenerowany JavaDoc (jako html). Opisuje on wszystkie zaimplementowane klasy wraz z metodami I polami. Normalnie w Javadocu uwzględnia się tylko pola I metody publiczne, ale my na potrzeby projektu zawarliśmy wszystkie.