

**Технически Университет – София**

**Факултет Приложна Математика и Информатика**

**Катедра Информатика**

**КУРСОВ ПРОЕКТ**

**Тема: Система за превенция на разпространение на вируси**

**Имена на студентите:** Станислав Бисеров Стоянов, Георги Донков Донков

**Факултетни номера, групи:** 471218066, 471218026, 76 група, 78 група

**Съдържание:**

1. Основни цели на архитектурата - описание на няколко ключови изисквания и ограничения на разработвания софтуерен продукт, които оказват влияние на избраната архитектура
2. Описание на основни сценарии (use-case) и актьори (actors), които са важни за разработваната архитектура, чрез use-case диаграми
3. Описание на логическия изглед на архитектурата. Клас диаграми за илюстрация връзките между архитектурно значими класове, подсистеми, пакети и слоеве. Диаграми на състоянието за илюстрация конкретни процеси на определени роли.
4. Изглед на процесите. Описание на отделните изпълними процеси (подсистеми) и зависимостите между тях.
5. Изглед на внедряването (Deployment view)
6. Изглед за разработка (Implementation view)
7. Обосновка на това как избраната архитектурата осигурява адекватна реализация на поставените нефункционални изисквания
8. Използвани източници

*1. Основни цели на архитектурата - описание на няколко ключови изисквания и ограничения на разработвания софтуерен продукт, които оказват влияние на избраната архитектура (Станислав Стоянов)*

Системата за превенция и разпрострение на вируси, за по-кратко навсякъде ще бъде използвана абревиатурата (VPS – Virus Prevention System) предоставя възможност за регистрация на потребители, такива, които имат потенциални симптоми или се връщат от държави с висока заболеваемост да бъдат лесно идентифицирани и открити от държавните власти и организации. Налице е също така подробна информация за всеки един потребител относно конкретните симптоми като медицинският персонал на болниците може чрез мобилното приложение на системата да назначава карантина с определен срок, базово лечение под домашни условия и наблюдаване текущото състояние на потребителя (пациента). Системата има уеб базиран потребителски интерфейс като избраните архитектурни стилове ще бъдат MVC ([Model-View-Controller](https://bg.wikipedia.org/wiki/Model-View-Controller)) и Microservices[[1]](#footnote-1), които ще разрешат евентуални бъдещи проблеми свързани с производителността (адекватна обработка на множество потребителски заявки без прeкъсване или отказ от страна на сървърите), скалируемостта и ефективността на системата. Системата ще предоставя и удобно безплатно мобилно приложение, достъпно в Google Play Store и App Store, работещо както за Android устройства, така и за IOS.

* Ключови изисквания

Основните ключови изисквания (нефункционални), които VPS трябва да има, са спазване на атрибутите за качество, много добре дефиниран дизайн, ориентиран към потребителя и спазване на различни методи за проектиране и използване на потребителския интерфейс (Usability Heuristics for User Interface Design[[2]](#footnote-2)). На първо място от съществено значение е да бъде взето под внимание времето за реакция на системата (responsiveness), защото това гарантира, че потребителят ще изживее максимално добри чувства ([Emotional design](https://www.interaction-design.org/literature/topics/emotional-design)) докато ползва системата. Дългите закъснения при обработка на заявките трябва да бъдат елиминирани. На следващо място идва сигурността (security), която гарантира, че сензитивните потребителски данни ще бъдат обработвани и използвани само от държавните служби. Цялата биологическа информация за потребителя не трябва да бъде публично достъпна и използвана без неговото разрешение. На последно място, но не по степен на важност е надеждността на системата (reliability). Този атрибут за качество гарантира, че средното време между евентуални откази (Mean Time Between Failure MTBF[[3]](#footnote-3)) на системата ще бъде максимално голямо, т.е VPS ще поддържа добри нива на безпроблемна работа, без особено значение какво натоварване ще бъде осъществено.

Добре дефинираният потребителски ориентиран дизайн и спазването на различни методи за проектиране и използване на потребителския интерфейс като: видимост на състоянието на системата, съвпадение между системата и реалния свят, постоянство, гъвкавост и ефективност на използването и евентуално възстановяване след грешка, причинена от потребителя са следващите ключови изисквания, които VPS трябва да реализира. За да може атрибутите за качество от бизнес гледна точка да бъдат спазени, то трябва много детайлно да бъдат решени въпроси като това дали системата изпраща необходимата обратна връзка на своите потребители (пациенти) под формата на различни известявания (email, лични съобщения към мобилния телефон, сесийни разговори с лекуващите лекари), дали има съвпадение между очакванията на потребителя за функционалността и реалната такава. На първо място, не е коректно, когато потребителят очаква да получи разрешение за излизане от карантина в конкретен ден да се получава забавяне и липса на известие или обаждане от гласов оператор. Особено значим момент е и системата да информира достатъчно добре потребителя за евентуални промени в предписаното лечение и диагноза на болестта. Постоянството от своя страна представлява потребителят бързо да се ориентира в използването на приложението и да не се чуди кой бутон да натисне или коя страница да отвори, а това се постига единствено със следване на определени конвенции от страна на разработчиците (спазване на КПК – Качество на Програмния Код, SOLID принципи и правилно наименуване на класове и методи). При настъпване на евентуални грешки във VPS, потребителят трябва да има възможност за изпращане на контактна форма с конкретния проблем или за по-добро потребителско изживяване Live chat, където да може конкретно да си зададе въпроса, а от другата страна човек по поддръжка на системата да съдейства максимално бързо. Всички горе изброени ключови изисквания трябва да бъдат взети под налбюдение, като разбира се, ако разработчиците изберат да следват agile методологиите, то във всеки един момент може да се променят или допълнят ключовите изисквания и фактори за развитие на системата.

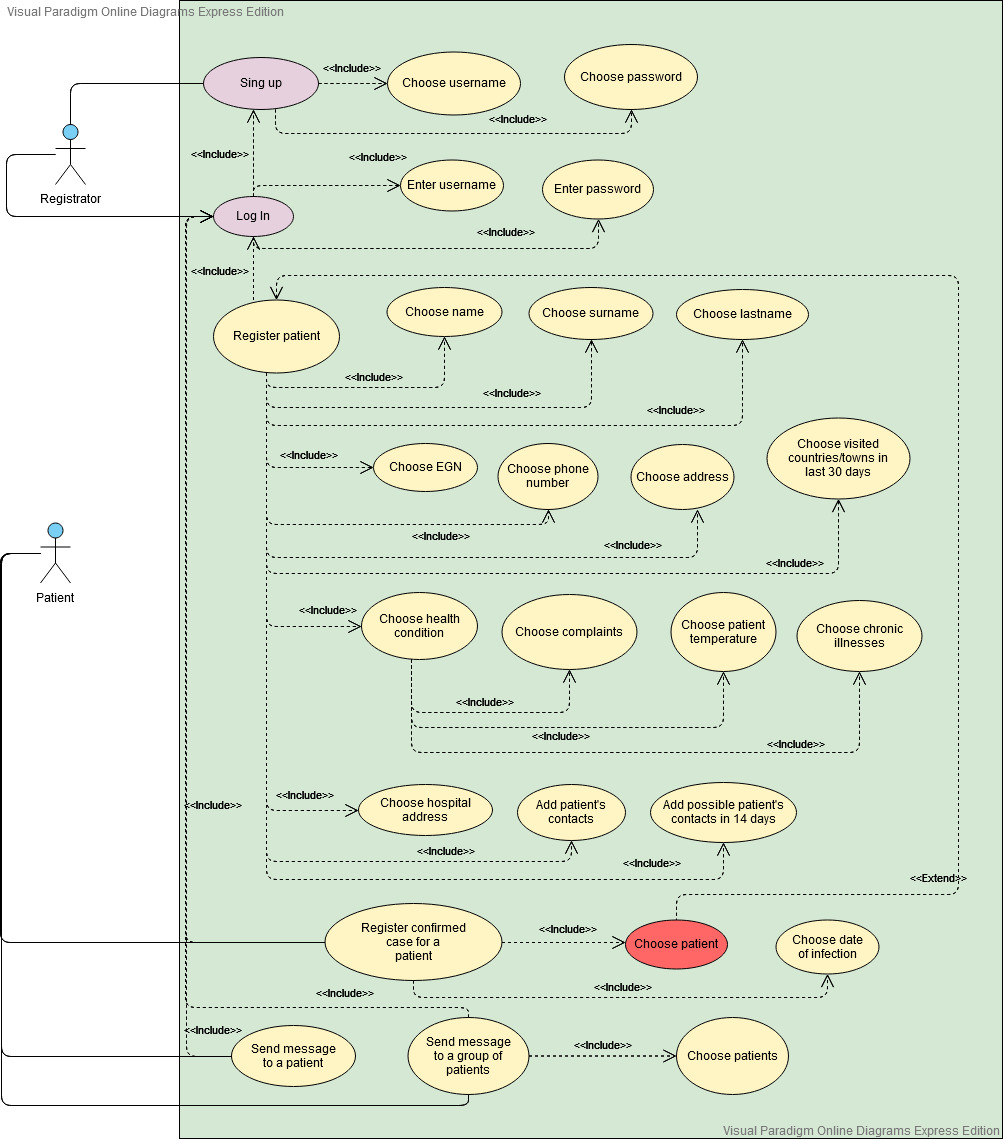
* Ограничения

Основното ограничение за реализация на системата е финансовият фактор. Преди създаването и имплементацията на всички изброени ключови изисквания, трябва човек от екипа, отговарящ за финансите, да разпредели максимално добре евентуалните бъдещи финансови средства. От атрибутите за качество, тук ясно изразен фактор е съвместимостта на системата (compatiblity), т.е до колко добре системата позволява оперативна съвместимост с по-стара наследена система. Този атрибут може да бъде пренебрегнат с цел намаляване на разходите, като в това число влиза и дизайнът на VPS. По-добре е да се отделят повече средства за производителност и надеждност, както и хардуерен ъпгрейд, отколкото за много красиво изглеждащо приложение. Друго ограничение е и тестваемостта на системата и до колко добре тя ще бъде покрита с необходимите стрес/load[[4]](#footnote-4) и regression[[5]](#footnote-5) тестове. Трябва да бъдат взети под внимание само тестовете свързани с производителността и надеждността, като целта е да бъде изтествана основната функционалност (при MVC влизат сървисите, отговарящи за бизнес логиката и контролерите). И последното съществено ограничение е времето за реализация на цялата система, защото компанията или държавната институция, която заявява създаването на системата налага и определени срокове за реализация. Представете си, че в държавата има епидемия от дадена болест (в днешни дни Covid-19) и се налага бързо създаване на приложение, което да бъде достъпно от всякакви устройства, с цел мониторинг, анализ и превенция на болестта. Ако системата ни бъде бързо проектирана и създадена, то шансовете ни за спечелване на търга и продажба на VPS са много по-големи.

*2. Описание на основни сценарии (use-case) и актьори (actors), които са важни за разработваната архитектура, чрез use-case диаграми*

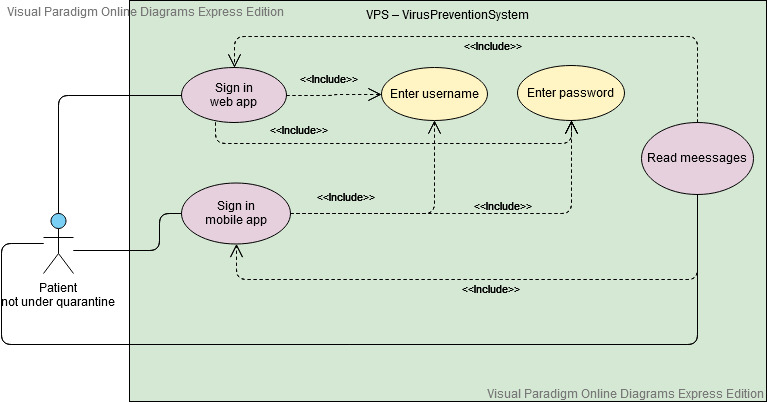
Чрез следните use-case диаграми представяме съществено важната функционалност на основните роли в системата.

* Регистратор *(Георги Донков)*



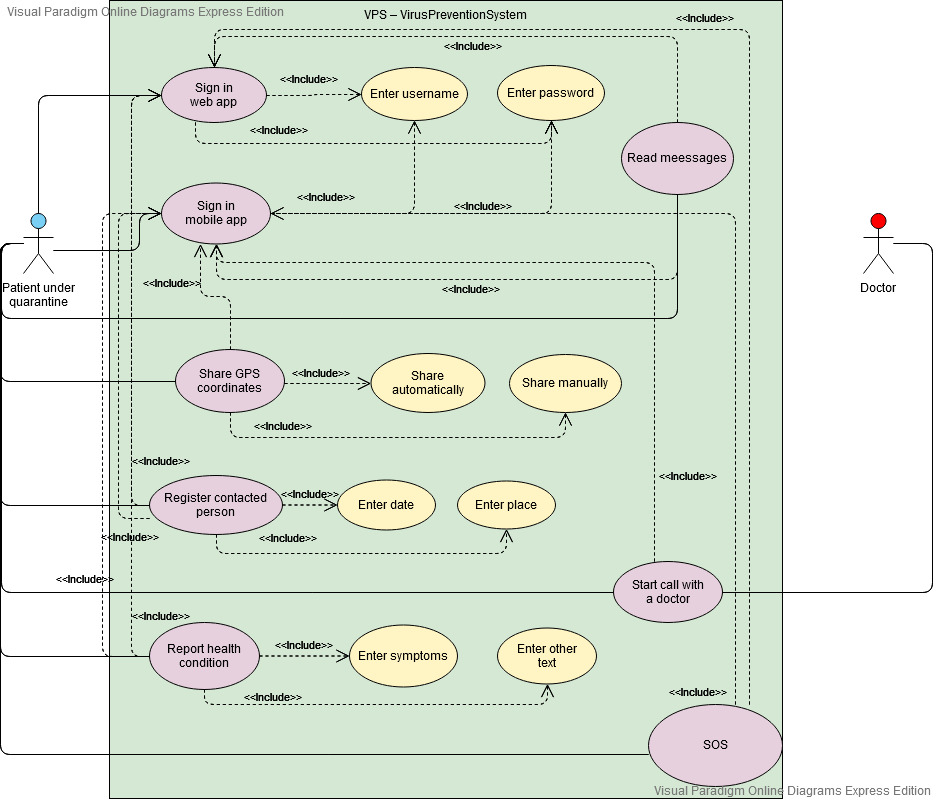
Регистраторът може да се регистрира като избере своя username и своята парола, с които после може да влезе в своя акаунт както в уеб-базираната система, така и в мобилната апликация. След влизането в системата регистраторът ще може да регистрира даден пациент, като въведе неговото име, презиме, фамилия, ЕГН, телефонен номер, адрес, посетени страни/градове през последните 30 дни, здравно състояние (включващо въвеждането на оплакванията на пациента, наличие на температура и списък от хронични заболявания), адрес на съответното за пациента болнично завдение, в което той ще бъде поставен под карантина, лицата, с които пациента е имал контакт през последните 14 дни и възможните лица, с които пациентът би имал контакт в бъдещите 14 дни. Регистраторът също така може да регистрира потвърдени случаи на зараза за съответни пациенти, като селектира заразилия се пациент и въведе датата на заразяване. Регистраторът ще може да изпраща съобщения до даден пациент или до група от пациенти, като преди изпращането на съобщението той ще трябва да селектира пациентите, до които иска да го изпрати.

* Пациент, който не е поставен под карантина *(Георги Донков)*



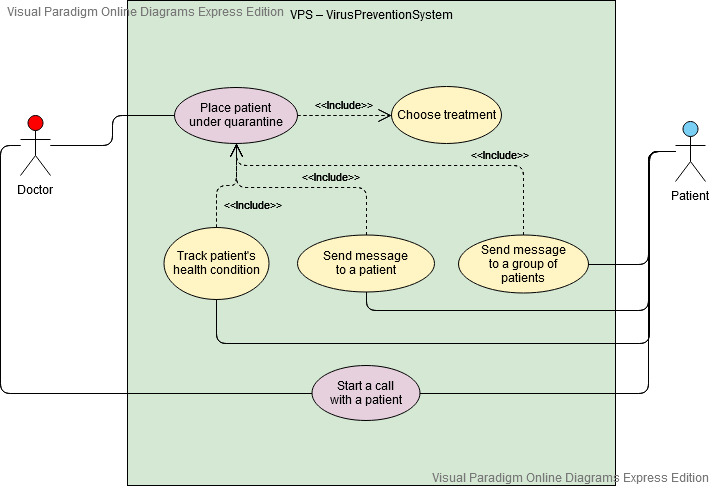
Пациентът, който не е поставен под карантина ще има възможност да влиза в своя акаунт в уеб базираната система или мобилното приложение, направен от регистратора, чрез въвеждането на username-а и паролата си. След влизането в акаунта си в някоя от двете системи, той ще може да чете изпратените до него съобщения.

* Пациент, който е поставен под карантина *(Георги Донков)*



Пациентът, който е поставен под карантина ще има възможност да влиза в своя акаунт в уеб базираната система или мобилното приложение, направен от регистратора, чрез въвеждането на username-а и паролата си. След влизането в акаунта си в някоя от двете системи, той ще може да чете изпратените до него съобщения. Пациентът ще може да споделя своите координати чрез GPS само в мобилното приложение, като избере някоя от двете опции са споделяне: ръчно или автоматично споделяне. Ако по време на карантината обстоятелствата са наложили пациентът да контактува с дадено лице, то за поставения под карантина се предоставя възможност да регистрира контактното лице, като въведе датата и мястото на контакт. Пациентът ще има възможността да докладва ежедневно здравното си състояние, като въвежда симптомите си или предоставя някаква допълнителна информация относно състоянието си в някое от двете приложения. Той също така ще може да разговаря с даден доктор само през мобилното приложение и ще може да изпраща SOS сигнали към лекарите при някаква извънредна ситуация какво през мобилното, така и през уеб приложението.

* Доктор *(Георги Донков)*



Докторите в системата ще имат възможността да поставят дадени пациенти под карантина, което ще включва избора на лечение за съответния пациент. Докторите ще могат да проследяват ежедневно здравното състояние на пациентите, поставени под карантина, да им изпращат набор от съобщения (групово или индивидуално) и също така при нужда да започнат разговор с даден пациент чрез някое от двете приложения.

* Служител в координационен център *(Станислав Стоянов)*

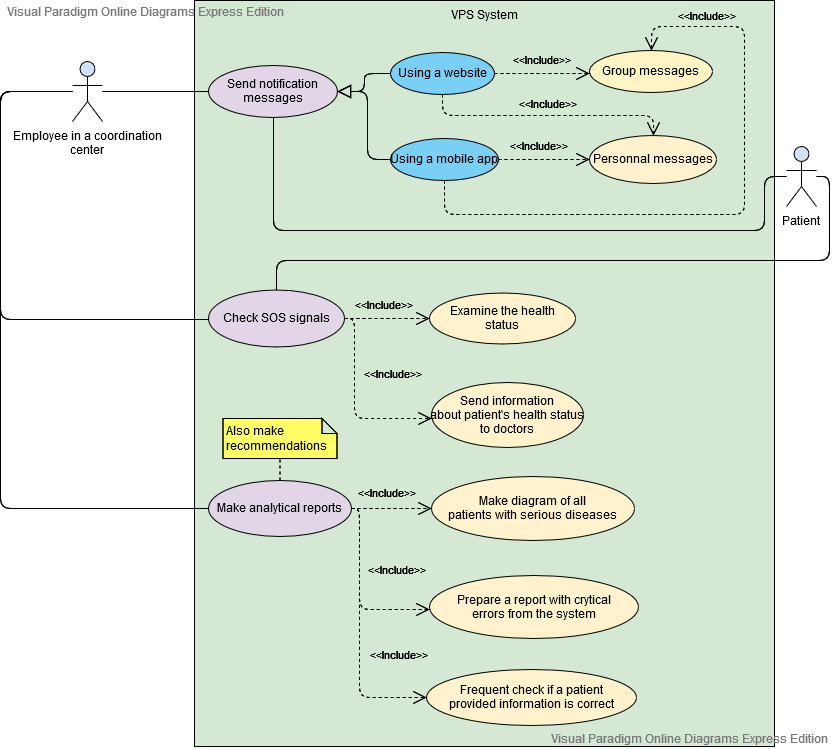
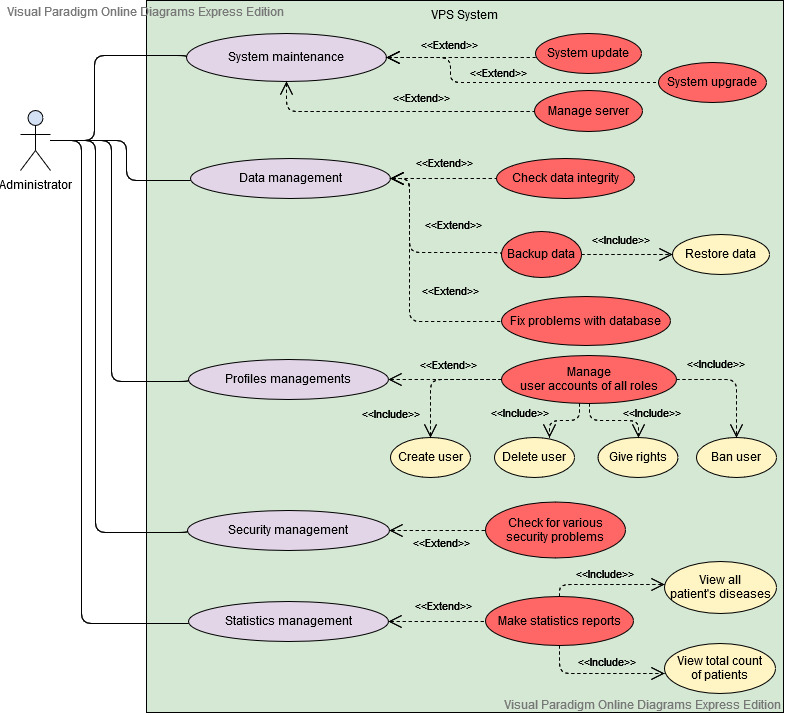


Figure 1- Employee in a coordination center Use-case diagram

Служителят в координационния център може да изпраща различни индивидуални или групови съобщения към поставените под карантина лица, като това може да се направи през мобилното приложение или уеб базирания интерфейс. Също така може да проследява SOS сигнали, изпратени от пациенти с влошаващо се състояние, включително да обработва тази информация и да я изпраща към съответните доктори. В заключение служителят може да прави аналитични доклади, да проследява всички процеси в системата и на база получената информация да изработва различни справки – диаграма на пациентите в тежко състояние, доклад за критичните грешки в системата и чести проверки коректността на потребителската биологична информация.

* Администратор *(Станислав Стоянов)*



Администраторът притежава множество функционалности като основните са: поддръжка на системата, поддръжка на Persistence layer (слоят, отговарящ за базата данни), управление на различните роли в системата, поддръжка на сигурността, изработка и оценка на статистически данни.

* Поддръжка на системата

Администраторът има права да извършва системни ъпдейти по график, да управлява функционалността на сървъра и да прави системни ъпрейди (увеличаване мощността на сървърните компоненти).

* Поддръжка на Persistence layer

Налице е поддръжка на данните, осъществяване на бекъп и възстановяване на системата от критични точки както и поправка на различни заявки в базата данни.

* Управление на ролите

Администраторът има права за създаване на потребител, изтриване и забраняване пълен или частичен достъп към системата. Също така може да предоставя права на останалите роли във VPS.

* Поддръжка на сигурността

Тук основната функционалност е в това, че администраторът може да проверява и разрешава проблеми със сигурността, в това число влизат: изтичане на потребителска информация или предоставянето на такава без разрешението на потребителя (в сила са и новите GDPR закони[[6]](#footnote-6)).

* Изработка и оценка на статистически данни

Администраторът може да изготвя различни статистически протоколи, да оценява разпространението на дадена болест и да преглежда количеството регистрирани потребители в системата.

*3. Описание на логическия изглед на архитектурата. Клас диаграми за илюстрация връзките между архитектурно значими класове, подсистеми, пакети и слоеве. Диаграми на състоянието за илюстрация конкретни процеси на определени роли.*

* Клас диаграми

*Вариант 1 (Станислав Стоянов)*

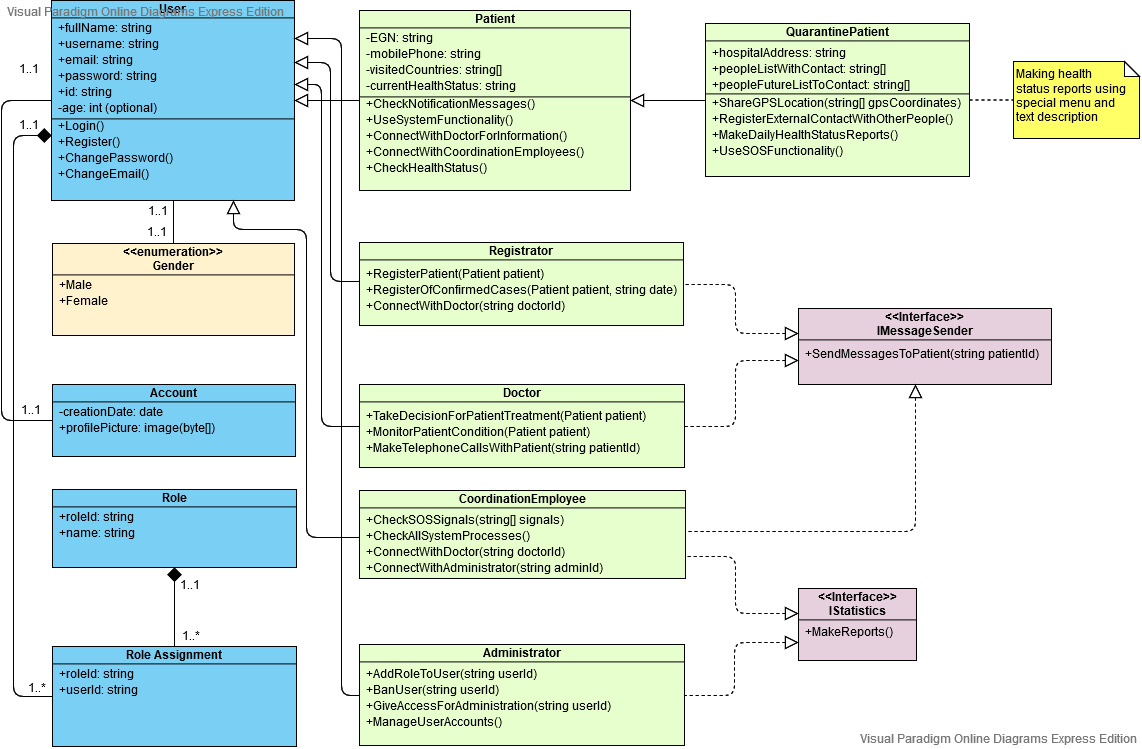


Figure 2- VPS class diagram

Вариант 1 клас диаграмата показва основните взаимовръзки между класовете в системата. Основните класове са следните: User, Account, Role, Role Assignment, Patient, Registrator, Doctor, CoordinationEmployee (служител в координационния център), Administrator, QuarantinePatient (пациент под карантина). Налице са и два интерфейса, които предоставят обща бизнес логика.

* Клас User

Класът User е базов за системата и съдържа основните полета за всеки един потребител в системата. Също така потребителят може да се логва, регистрира, променя своята парола или имейл. Задължително всеки потребител трябва да заяви своя пол посредством енумерацията “Gender” с две опции – Male и Female.

* Клас Account

Всеки потребител във VPS може да има само един акаунт, като информацията, която съдържа акаунтът е профилната снимка на потребителя и датата на създаване.

* Класове Role и RoleAssignment

Ролята съдържа уникално ид и име, като тя е в композиция с RoleAssignment, т.е класът RoleAssignment има за цел да бъде свързващ клас между потребителя и конкретната роля. Ограничението тук е, че не може да съществува роля без RoleAssignment и също така един потребител може да има повече от една роля. Целта на тези класове е ясно да се разграничат правата, които всеки потребител може да има в системата.

* Клас Patient

Класът Patient съдържа основната информация за един пациент. Този клас е наследник на базовия клас User, т.е получава функционалността на един потребител. Също така пациентът може да следи своите съобщения и известия, да използва основната функционалност на системата, да се свързва с лекуващия доктор и служителите в координационния център както и да проверя статуса на текущото си здравословно състояние.

* Клас QuarantinePatient

Класът QuarantinePatient наследява своя базов клас Patient и User. Като такъв той получава функционалността на тези класове и представя поведението на пациент под карантина. Може да споделя текущото си местоположение чрез мобилното приложение на системата, да регистрира списък с хората, с които е имало продължителен контакт, да докладва ежедневно здравословното си състояние и да използва SOS функционалността.

* Клас Registrator

Регистраторът е базов клас на User и наследява интерфейса IMessageSender, тоест може да изпраща персонални или групови съобщения на даден пациент. Също така регистраторът прави регистрация на лица (пациенти), потенциални обекти на зараза. Той може да регистрира и потвърдени случаи на заразяване на пациенти и да осъществява конткат с лекуващия доктор.

* Клас Доктор

Докторът отново е базов клас на User и наследява интерфейса IMessageSender. Той може да назначава лечение, диагноза на конкретен пациент и да взема решения за поставяне на дадено лице под карантина. Също така може да проследява състоянието на поставен под карантина пациент и да провежда разговори.

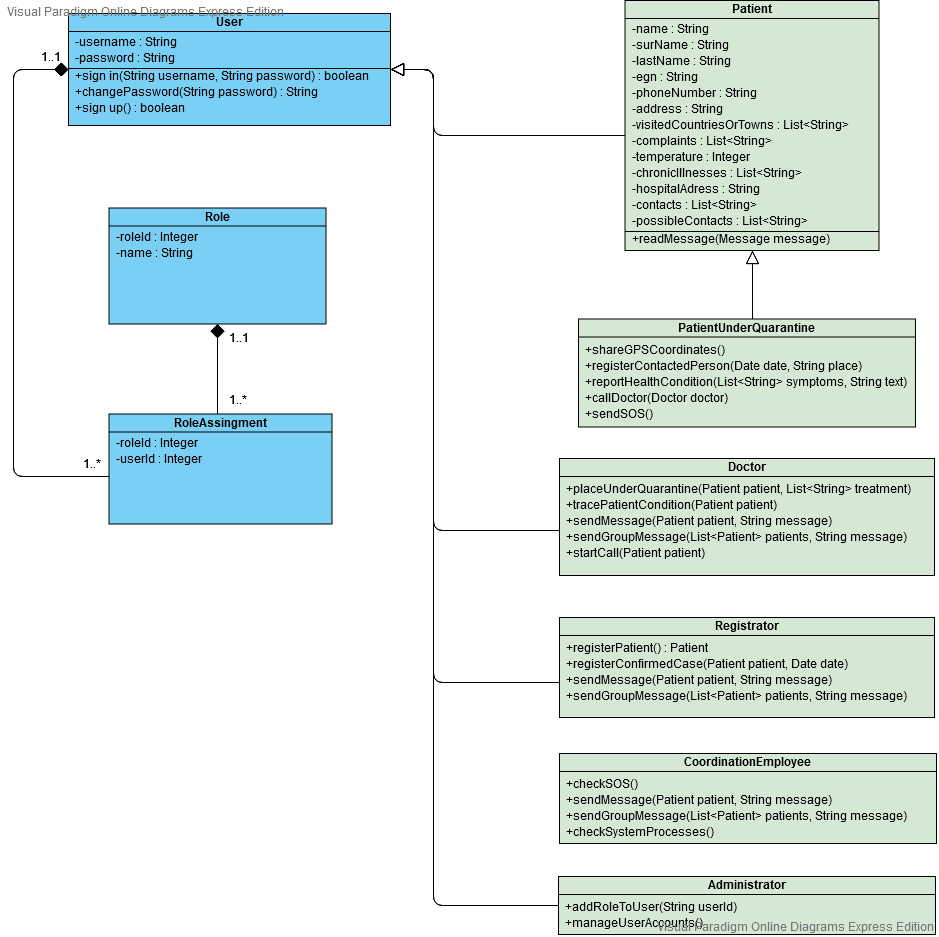
* Клас CoordinationEmployee

Служителят в координационния център е базов клас на User, наследява интерфейса за изпращане на съобщения както и интерфейса IStatistics, с което получава функционалност за правене на доклади и анализи. Той може да проследява SOS сигнали от поставените под карантина лица, да следи всички процеси в системата, да се свързва с конкретен лекар на пациента и да осъществява контакт с администратора на VPS.

* Клас Administrator

Администраторът има пълен достъп до цялата система. Може да управлява всички потребителски акаунти и роли, да дава достъп на определени потребители, да прави доклади, благодарение на това, че наледява интерфейса IStatistics и да налага забрани за ползване на системата.

*Вариант 2 (Георги Донков)*



Вариант 2 клас диаграмата показва друга възможност на основните взаимовръзки между класовете в системата. Основните класове са следните: User, Role, RoleAssingment, Parient, PatientUnderQuarantine, Doctor, Registrator, CoordinationEmployee, Administrator.

* Клас User

Класът User е базов за системата и съдържа основните полета за всеки един потребител в системата(username, password). Всеки обект от класа User ще може да влиза в системата и да променя своята парола. Методът за регистрация в системата ще бъде абстрактен, защото един пациент не може да се регистрира, а може само да влиза в системата (неговата регистрация се извършва от даден регистратор).

* Класове Role и RoleAssignment

Ролята съдържа уникално ид и име, като тя е в композиция с RoleAssignment, т.е класът RoleAssignment има за цел да бъде свързващ клас между потребителя и конкретната роля. Ограничението тук е, че не може да съществува роля без RoleAssignment и също така един потребител може да има повече от една роля. Целта на тези класове е ясно да се разграничат правата, които всеки потребител може да има в системата.

* Клас Patient

Класът Patient съдържа основната информация за един пациент. Този клас е наследник на базовия клас User, т.е получава функционалността на един потребител. Пациентът, който не е поставен под карантина, може единствено да прочита съобщенията, изпратени към него от докторите, регистраторите или слуцителите в координационния център чрез метода readMessage().

* Клас PatientUnderQuarantine

Класът PatientUnderQuarantine наследява класовете Patient и User. Като такъв той получава функционалността на тези класове и представя поведението на пациент под карантина. Един пациент под карантина може да споделя своето GPS местоположение чрез метода shareGPSCoordinates, да регистрира контакти с него лица, като предоставя датата и мястото на контакт чрез метода registerContactedPerson, да докладва ежедневно здравното си състояние чрез метода reportHelathCondition, като спомене своите симптоми и добави допълнителна информация относно състоянието си(опционално), да се обажда на определен доктор чрез метода callDoctor и да изпраща SOS сигнали чрез метода sendSOS.

* Клас Registrator

Класът регистратор наследява класа User и като такъв той получава неговата функционалност. Един регистратор може да регистрира пациенти чрез метода registerPatient, да регистрира потвърдени случаи на зараза чрез метода registerConfirmedCase, предоставяйки конкретния заразен пациент и датата на зараза, да изпраща съобщения до пациент чрез метода sendMessage или да изпраща съобщения до група от пациенти чрез метода sendGroupMessage.

* Клас Доктор

Класът доктор наследява класа User и като такъв той получава неговата функционалност. Един доктор може да поставя пациенти под карантина чрез метода placeUnderQuarantine, да проследява здравното състояние на даден пациент чрез метода tracePatientCondition, да изпраща съобщение до пациент чрез метода sendMessage, да изпраща съобщение до група от пациенти чрез метода sendGroupMessage и да започне разговор с даден пациент чрез метода startCall.

* Клас CoordinationEmployee

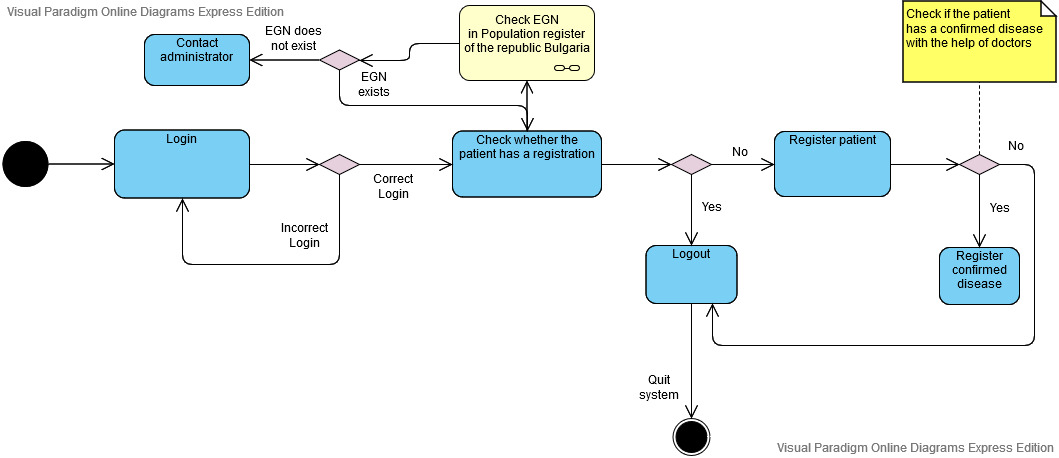
Класът CoordinationEmployee наследява класа User и като такъв той получава неговата функционалност. Всеки обект от този клас може да проследява SOS сигнали на поставени под карантина пациенти чрез метода checkSOS, да изпраща съобщение до пациент чрез метода sendMessage, да изпраща съобщение до група от пациенти чрез метода sendGroupMessage и да проверява системните процеси чрез метода checkSystemProcesses.

* Клас Administrator

Администраторът има пълен достъп до цялата система. Обект от този клас може да дава роли на абсолютно всички потребители в системата и да управлява акаунтите на потребителите.

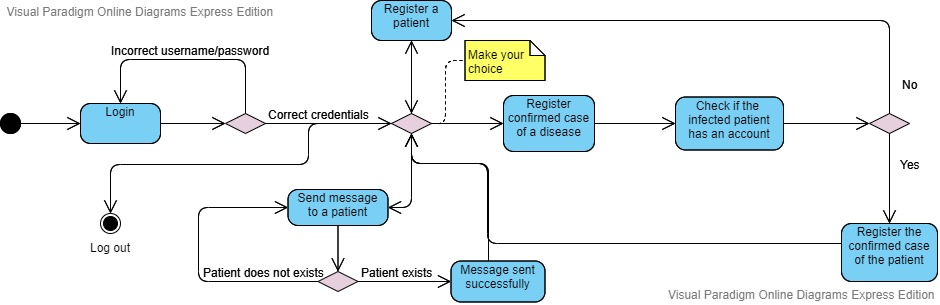
* State diagram (диаграма на състоянието) за конкретен процес

*Вариант 1 (Станислав Стоянов)*



На следната диаграма на състоянието е представен процесът при регистрация на пациент от регистратора, като това включва проверка дали вече пациентът е регистриран в системата с валидно ЕГН. Ако не съществува такова ЕГН, регистраторът трябва да се свърже с администратора на VPS, а ако е валидно се преминава към стъпката за регистрация. При нея се проверява дали пациентът има потвърдена болест от докторите и ако има се продължава към регистрация на потвърден случай на заразяване. Накрая след като се премине през описаните процеси, регистраторът излиза от системата.

*Вариант 2 (Георги Донков)*

**

На следната диаграма на състоянието са представени дейностите, които един регистратор може да извършва в системата. Всеки регистратор като за начало трябва да влезе в своя акаунт в системата. Ако влизането му е успешно, то той може да извършва набор от дейности. Регистраторът може да регистрира нови пациенти в системата. Също така той може да регистрира потвърдени случаи на определено заболяване на пациент (ако пациентът все още не е регистриран в системата, то регистраторът трябва да го регистрира). Регистраторът също така има възможността да изпраща съобщения до пациенти.

*4. Изглед на процесите. Описание на отделните изпълними процеси (подсистеми) и зависимостите между тях.*

Представяме изгледа чрез две възможности на реализация на UML диаграмите за последователност (Sequence diagrams) и активност (Activity diagrams).

* Диаграми за последователност

*Вариант 1 (Станислав Стоянов)*

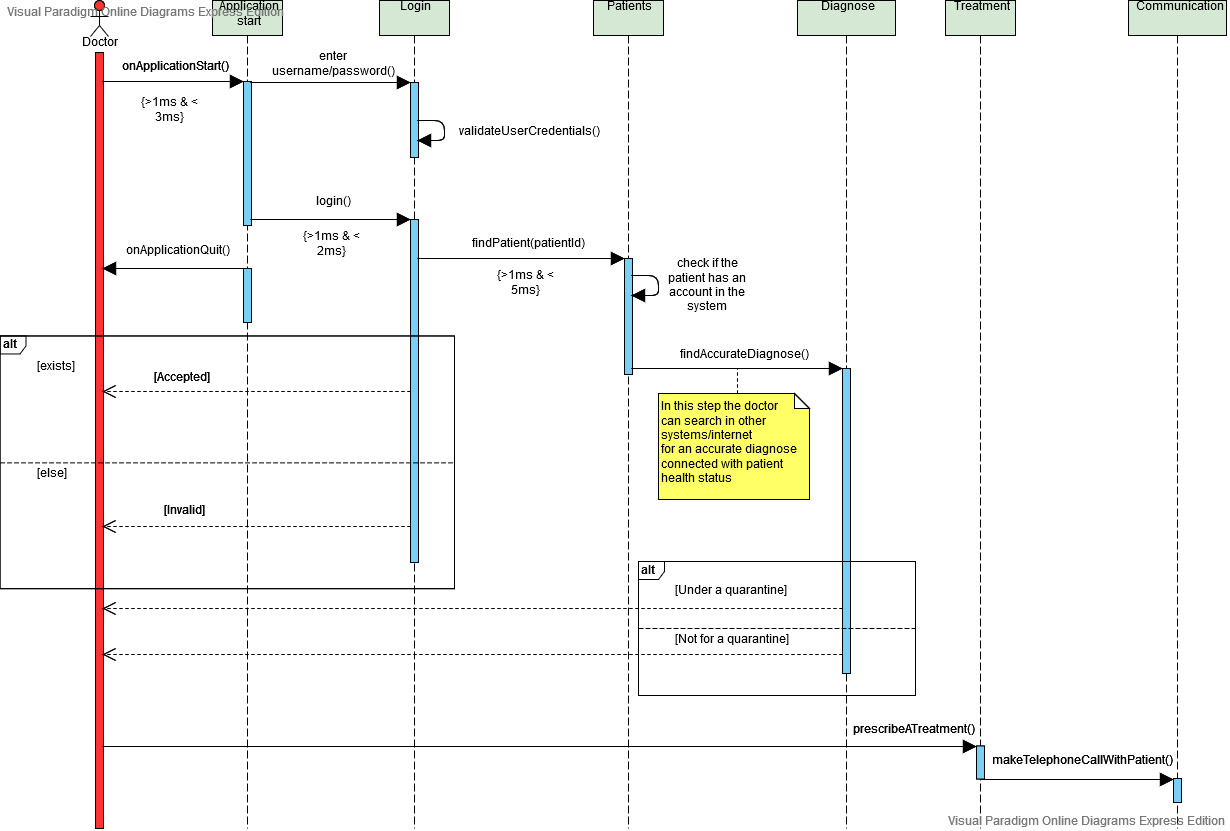
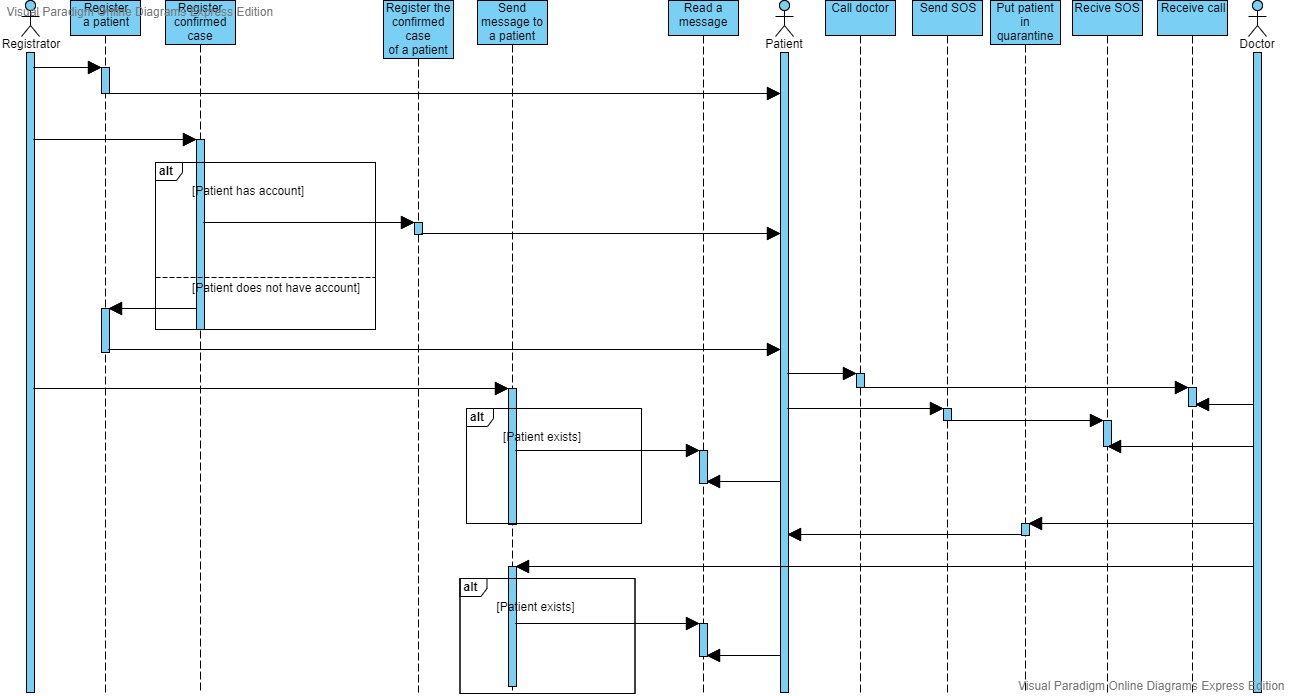
**

Figure 3- Sequence diagram for a doctor process

На следната диаграма на последователност е изобразен целият процес за вземане на решение относно поставяне на лице/пациент под карантина и предписване на последващо лечение от назначения доктор. Какво се включва в целия този процес? На първо място докторът трябва успешно да се идентифицира със своето потребителско име и парола, след това да намери съответния пациент в списъка с всички пациенти (това включва и проверка дали изобщо има такъв пациент в системата) и да постави точна диагноза, позовавайки се на знанията си или чрез помощта на други системи/интернет. После трябва да реши дали да постави пациента под карантина или не, да предпише лечение и накрая да проведе телефонен разговор с конкретния пациент.

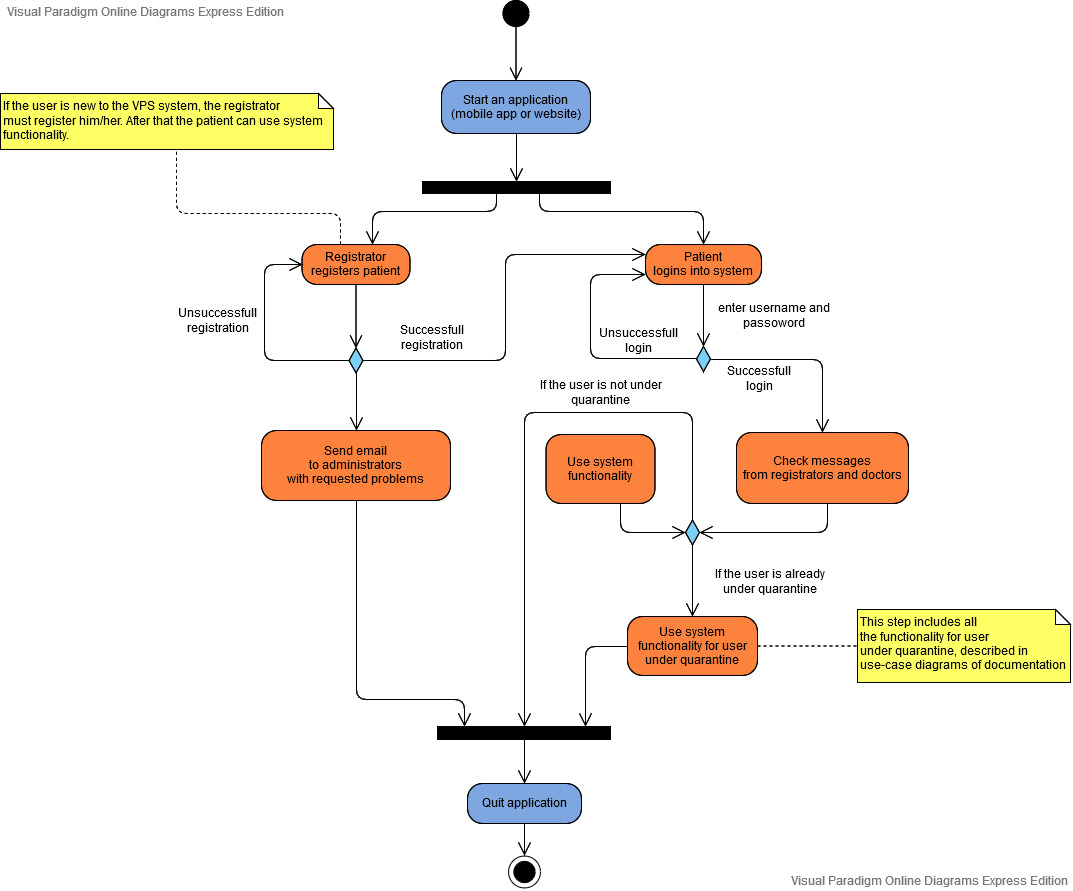
*Вариант 2 (Георги Донков)*



На следната диаграма на последователност са изброени дейностите, които могат да се извършват от трите роли, част от нашата софтуерна система. Спестени са малки детайли в диаграмата, както и останалите две роли, с оглед на това да се събере в образеца за курсов проект.

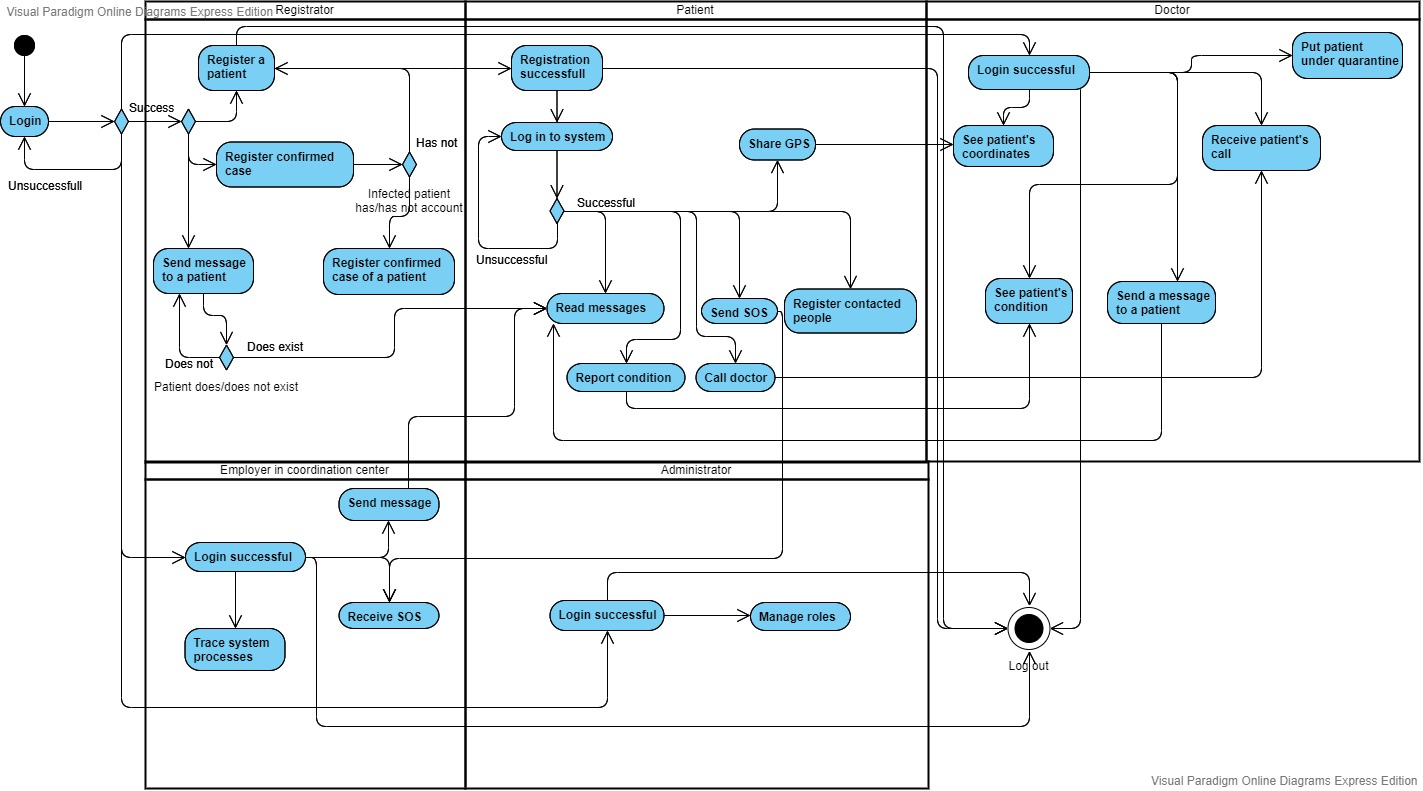
* Диаграми за активност

*Вариант 1 (Станислав Стоянов)*



На показаната диаграма за активност са изобразени процесите и функционалността на един пациент, ползващ системата. Първоначално задължително условие е той да има регистрация, направена от регистратора на системата и да се е успешно логнал (при евентуални грешки от страна на системата, пациентът има възможност да се свърже с администратора и да докладва за проблема). След това пациентът може да провери всички изпратени съобщения от регистраторите и докторите, да използва основната му предоставена функционалност и конкретната такава, ако вече е поставен под карантина. След завършване на тези процеси, може да излезе от системата (logout).

*Вариант 2 (Георги Донков)*

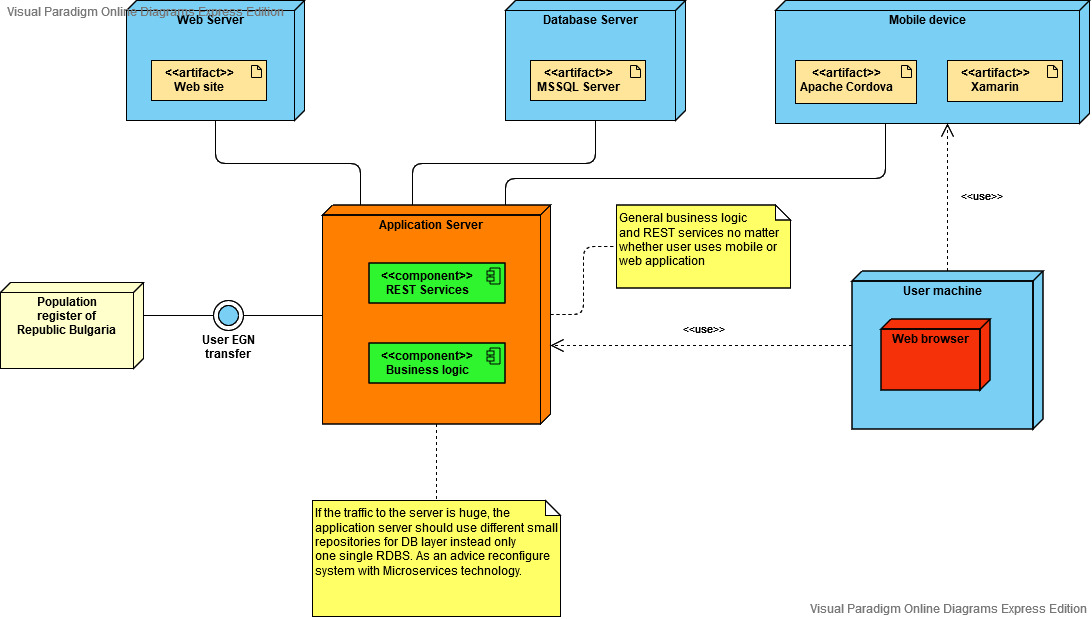
**

На следната диаграма на активност са показани и свързани дейностите, които могат да се извършват от всички роли в нашата софтуерна система. Отново са спестени дребни детайли в диаграмата с оглед да се събере в образеца за курсов проект.

*5. Изглед на внедряването (Deployment view)*

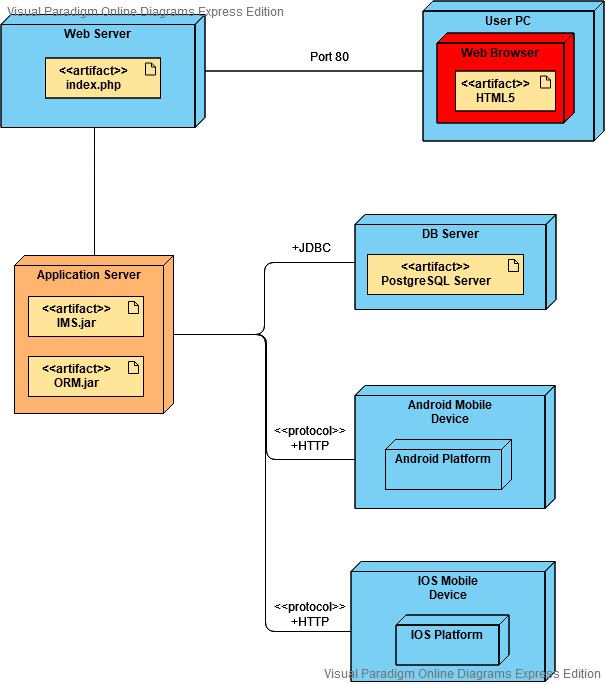
Този изглед е представен чрез две възможности на реализация на UML Deployment диаграми, което ще подпомогне инженерът да конфигурира по-бързо необходимите сървърни среди и средата, където ще се инсталира системата. Също така е описано разположението на отделните модули на VPS, тоест уеб сайтът за уеб достъп до системата, клиентското мобилно приложение и общата бизнес логиката на приложения сървър заедно с базата данни.

*Вариант 1 (Станислав Стоянов)*



На следната UML Deployment диаграма е представен основният физически изглед на системата и необходимите отделни компоненти. Понеже архитектурният стил на VPS е MVC с евентуално ползване на микросървиси, то тогава следва, че ще има уеб сървър, Persistence layer (в случая релационна база данни – MSSQL сървър) и Application сървър, който си комуникира с тях посредством REST услуги. Както уеб приложението/сайтът така и мобилното приложение ползват обща бизнес логика на този сървър. При увеличаване потребителските заявки и проблеми с времето за отговор (responsiveness), може да се премине към бъдеща имплементация на микросървъси и множество “малки” репозиторита, които значително да подобрят производителността на системата. В заключение трябва да се изгради интерфейс за комуникация между Application сървъра и Регистъра на населението на Република България, за да се обменя бързо информация като ЕГН-то на конкретния пациент.

*Вариант 2 (Георги Донков)*

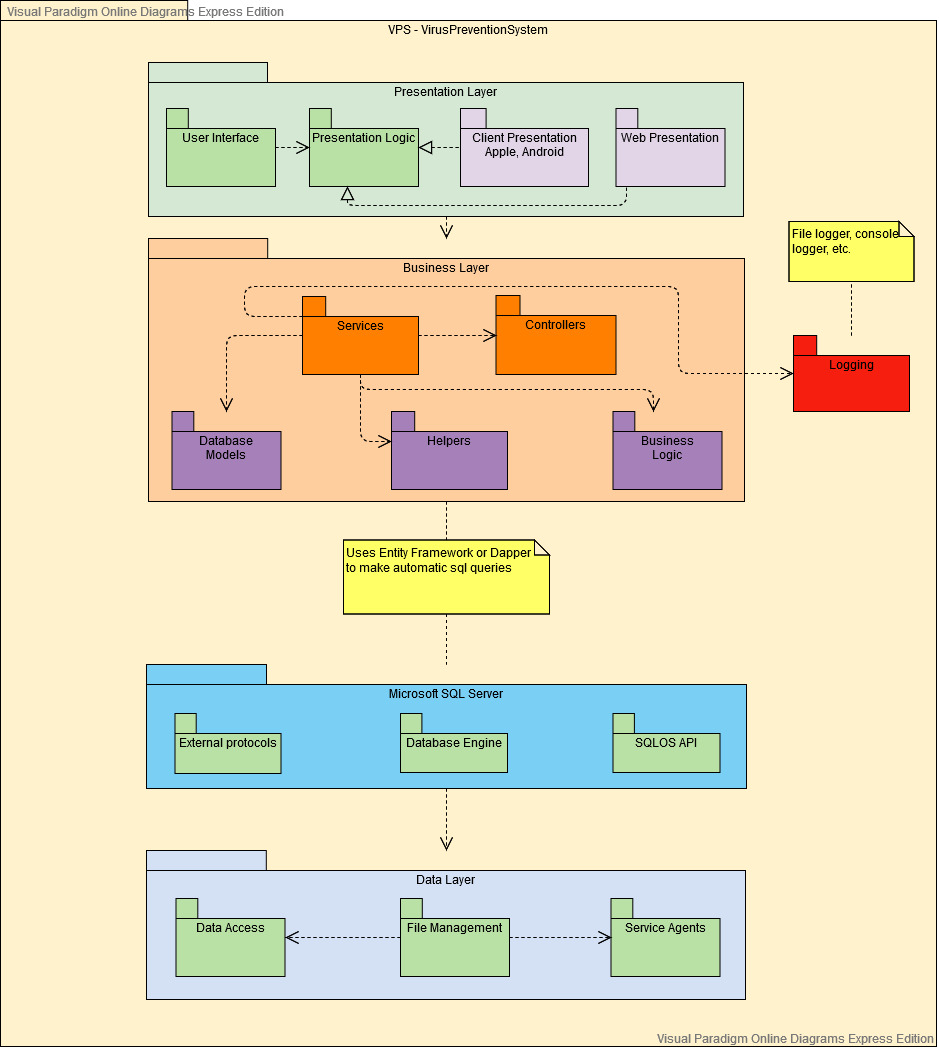


На следната UML Deployment диаграма е представен основният физически изглед на системата и необходимите отделни компоненти. Понеже нашият софтуер ще се състои от мобилно и уеб приложение, то следва, че трябва да имаме един Web Server, с който да комуникират нашите потребители посредством интернет браузъри на техните компютри, един Application Server, към който да е свързан въпросният Web Server, както и сървърът за базата данни и мобилните устройства на нашите потребители (в нашия случай ще предоставим приложение, което да е съвместимо както с Android, така и с IOS операционните системи). За базата данни можем да използваме PostgreSQL.

*6. Изглед на разработка (Implementation view)*

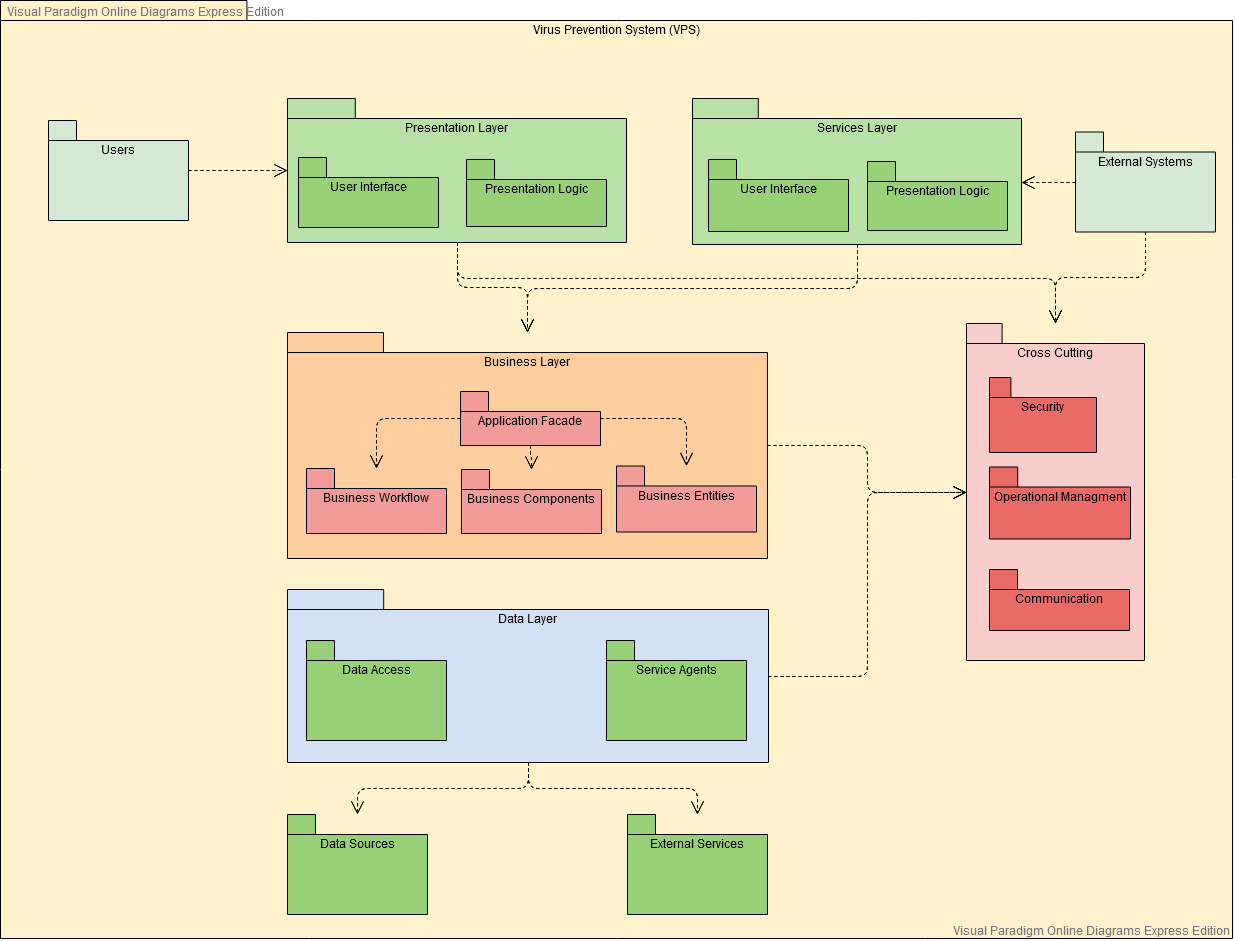
Изгледът на разработка описва основните класове на отделните модули на системата. За представянето му сме използвали два варианта на UML Package диаграми.

*Вариант 1 (Станислав Стоянов)*



Следната UML Package диаграма представя основните слоеве (пакети) на архитектурния стил MVC. Системата съдържа презентационен слой (Presentation layer), който от своя страна съдържа два изгледа – един за уеб приложението и неговия интерфейс и един за мобилното приложение. Презентационният слой комуникира с бизнес слоя (Business layer), тоест основната функционалност на приложението е съсредоточена в този слой, като според MVC патърна бизнес логиката се съдържа в сървисите, които работят с модели от базата данни, а контролерите ползват наготово тази логика (контролерите ползват специални ViewModels и чрез библиотеки като Automapper визуализират данните на UI слоя). Те не трябва да знаят за моделите в БД и пряко да може да променят тези данни. Сървисите от своя страна могат да работят директно с данните в БД, да използват хелпъри (Helpers) при нужда за определена логика и да записват логове в специални логъри (например в текстови файл или някаква конзола). Тези логъри могат съществено да помогнат на разработчиците при търсене на евентуални грешки или други проблеми със системата. Как всъщност се осъществява връзката с БД и директно ли се изпращат SQL заявки? С цел улеснение всяка една по-сериозна система ползва готов Framework (Entity Framework[[7]](#footnote-7), Dapper[[8]](#footnote-8), Hibernate[[9]](#footnote-9)), който прави лесно създаването на заявки към БД без да се пише SQL (под формата на LINQ, конкретно при реализация на системата със C#). След това SQL сървърът комуникира с ядрото на БД и прави необходимите заявки, транзакции или други queries. Отново както беше споменато и в т.5 от документацията, при проблеми с времето на реакция и забавяне от страна на БД, може да се прибегне към ползване на микросървисна архитектура.

*Вариант 2 (Георги Донков)*



Вариант 2 UML Package диаграмата представя основните слоеве на MVC архитектурния стил, който се състои от 3 основни пакета – презентационен слой (Presentation layer), Services layer, който е част от Business слоя и Data layer (БД). Потребителите комуникират главно с презентационния интерфейс, докато сървисите както е показано на диаграмата могат да бъдат преизползвани от външни системи. Cross Cutting е слой, който отговаря за сигурността, поддръжката на операциите и комуникацията вътре в бизнес логиката на системата. Слоят, отговарящ за данните единствено обменя информация с бизнес логиката. Груба грешка е потребителският интерфейс да има директна връзка с базата данни от архитектурна гледна точка. БД в случая е релационна, но в по-големите реални системи може да бъде и разпределена.

*7. Обосновка на това как избраната архитектурата осигурява адекватна реализация на поставените нефункционални изисквания (Георги Донков)*

Първото нефункционално изискване, което нашата софтуерна система трябва да покрива, е производителността, или по – точно системата ни да извършва обработка на потребителски заявки в нормално възприетото за това време (2 секунди), дори и при многократно нарастване на броя едновременно работещи с нея потребители. Това значи, че ние трябва да осигурим мощен сървър за нашата система, който да може да се справя с огромен наплив от потребителски заявки, без отговорите му да закъсняват с повече от 2-3 секунди. Всъщност производителността е пряко свързана с друго нефункционално изискване, което системата трябва да покрие, а именно скалируемостта – нашата система трябва да позволява лесно преконфигуриране на отделните модули при нарастване на обема на потребителски заявки с цел запазване на определената производителност. Добрата скалируемост в архитектурата се постига посредством микросървисната архитектура, която евентуално ще използваме. Благодарение на нея всеки един сървис може да се разглежда като една самостоятелна единица, която може да се изгради и след това да се коригира, без да се засяга работата на нито един от останалите сървиси. Именно по този начин ние осигуряваме лесното преконфигуриране на всеки модул при определена ситуация. Друго нефункционално изискване, което системата ни трябва да покрие е сигурността – VPS трябва да гарантира, че чувствителни данни няма да бъдат достъпни за трети лица. Това ще може да се постигне чрез строго определените роли в системата, които могат да се дават само от администратор. Пациентите ще могат да се регистрират единствено от регистраторите, които след регистрацията ще им предоставят данните, с които да могат да влизат в приложението (потребителско име и парола). Достъп до потребителските данни ще имат само регистраторите, докторите и администраторът, които също ще имат своите начини за идентификация в системата. Възможно решение, което ще подсигури сигурността на системата, е да се добави така нареченото two-factor authentication. Това може да се постигне, като изградим софтуера така, че при всяко влизане на потребител, независимо с каква роля, да се изпраща СМС или електронно съобщение с код, който да бъде нужен за успешното влизане. След влизането в системата от определено устройство, всеки един потребител ще има опцията да го запази, за да не се налага отново получаване на СМС/електронно съобщение. След запомняне на дадено устройство, код ще се изпраща и ще се очаква неговото въвеждане единствено и само ако даден потребител се опита да влезе в своя акаунт чрез друго, незапомнено до момента устройство. Последното нефункционално изискване, което системата ни трябва да покрива, е отказоустойчивостта, тоест системата ни да продължи да работи коректно, дори и при отпадане поради някаква авария на някои от основните сървъри. Това може да се постигне чрез закупуването на така наречения backup сървър, или на български второстепенен сървър, който да се включва само при възникването на дадена повреда в основния сървър, за да може да не се наруши работата на системата за неопределен период от време. За да се спестят пари може да се закупи по – слаб второстепенен сървър, който да може да поддържа работата на системата с прилични темпове, докато не се отстрани проблемът, възпрепятствал работата на основния сървър. Разбира се, по – евтиният второстепенен сървър е възможно да не реагира толкова бързо колкото основния в момент на пренатоварване и ако искаме да спестим парични средства, ние трябва да подсигурим високо квалифициран екип за поддръжка на сървърите, който да може да се справи възможно най – бързо с проблема като го отстрани и възвърне нормалното функциониране на софтуера.

*8. Използвани източници*

* лекции и презентации за UML диаграмите, архитектурни стилове и атрибути за качество от курса
* Даниел Митев, Дизайн, ориентиран към потребителя (User Centered Design). Основни етапи при проектирането на ДОКП
* Даниел Митев, Методи за проектиране и използване на потребителски интерфейс (Usability Heuristics for User Interface Design). Схематичен дизайн
* знания от предишни курсове (ООП, СУБД)
* за информацията под чертите са използвани препратки от Wikipedia

1. Микроуслугите се насочват в приложения, където свързването е възможно най-слабо (loose coupling) и кохезията е възможно най-силна (strong cohesion) – (smart endpoints and dumb pipes). [↑](#footnote-ref-1)
2. Основните методи за проектиране и използване на потребителски интерфейс са създадени през 1990г. от Джакопо Нилсен в колаборация с Ролф Молих. [↑](#footnote-ref-2)
3. MTBF = MTTF (Mean Time To Fail) + MTTR (Mean Time To Repair) [↑](#footnote-ref-3)
4. Стрес/Load тестовете представляват симулация на голям поток от потребителски заявки и гарантират надеждност и липса абнормалност на системата [↑](#footnote-ref-4)
5. Regression тестовете осигуряват това, че поправен бъг в системата няма да се повтори отново в друг компонент [↑](#footnote-ref-5)
6. GDPR (General Data Protection Regulation) е новият Общ регламент на Европейския съюз за защитата на данните (ОРЗД) от 27 април 2016 година, който регулира обработването, съхранението и използването на лични данни на физически лица от други лица, дружества или организации. [↑](#footnote-ref-6)
7. **Entity Framework** (**EF**) is an open source [object-relational mapping](https://en.wikipedia.org/wiki/Object-relational_mapping) (ORM) framework for [ADO.NET](https://en.wikipedia.org/wiki/ADO.NET). It was a part of [.NET Framework](https://en.wikipedia.org/wiki/.NET_Framework), but since Entity Framework version 6 it is separated from .NET framework. [↑](#footnote-ref-7)
8. **Dapper** is an [object-relational mapping](https://en.wikipedia.org/wiki/Object-relational_mapping) (ORM) product for the [Microsoft .NET](https://en.wikipedia.org/wiki/.NET_Framework) platform: it provides a [framework](https://en.wikipedia.org/wiki/Software_framework) for mapping an [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming) [domain model](https://en.wikipedia.org/wiki/Domain_model) to a traditional [relational database](https://en.wikipedia.org/wiki/Relational_database). [↑](#footnote-ref-8)
9. **Hibernate ORM** (or simply **Hibernate**) is an [object-relational mapping](https://en.wikipedia.org/wiki/Object-relational_mapping) tool for the [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) programming language. [↑](#footnote-ref-9)