Sprint Two:

Requirements Engineering:

Project Problem #7: You've been asked by the newly formed Rutgers Health to create a patient-doctor appointment system. Create a system that categorizes illness into N categories and doctor medical specialization into M categories (where N > M). Ideally, every patient will get a doctor with the right specialty but that could result in long waiting times. You will need to make necessary assumptions about the number of patients, the number of doctors, acceptable waiting times (say based upon illness type), etc. but no patient must remain unseen for more than 14 days under any conditions.

<u>Part A) User and System Requirements:</u>
User Requirements:

The patient-doctor appointment system shall create an appointment for a patient given their illness where no patient must wait more than 14 days to get an appointment. Each doctor should be able to view a list of all appointments that have been scheduled with them over the next two weeks.

System Requirements:

1. Using data on common illnesses suffered by patients, N general categories/types of illnesses will be created. The specific illness will be categorized into a general category that it is associated with.
2. Given knowledge of doctors and common medical specialties, M categories of medical specialties will be created, and N > M.
3. The system will be designed in such a way that those patients who are suffering more seriously and have emergencies will be given appointments first. In this manner, the system will run based on priority.
4. The system will be dependable and ensure that no inputs cause the system to fail
5. The system shall incorporate a service oriented architecture

<u>Part B) Functional & Non-Functional Requirements:</u>
Functional Requirements:

- A user (patient) shall be able to enter illness information into the system and receive appointment information (time, name of the doctor, address, etc.)
- A user (doctor) shall be able to enter their name and ID into the system and get a schedule of their appointments with their patients.
- The system shall create an appointment for every patient with a doctor with the right specialty.
- The system shall ensure that no patient must wait more than 14 days to get an appointment.
- A user (patient) shall be able to reschedule their appointment.

- The system shall ensure that each patient gets a unique appointment, i.e., no two patients must get the same appointment with the same doctor at the same time.
- The system shall ask the user if they want to view a list of out of the system doctors when the user declines the appointment.
- The system should be reliable - and ensure any entered input by the user does not cause the system to fail.
- The system should check for invalid inputs (wrong format inputs or inputs beyond the specified bound of inputs) before accessing the database with these inputs and ask the user to provide valid inputs.

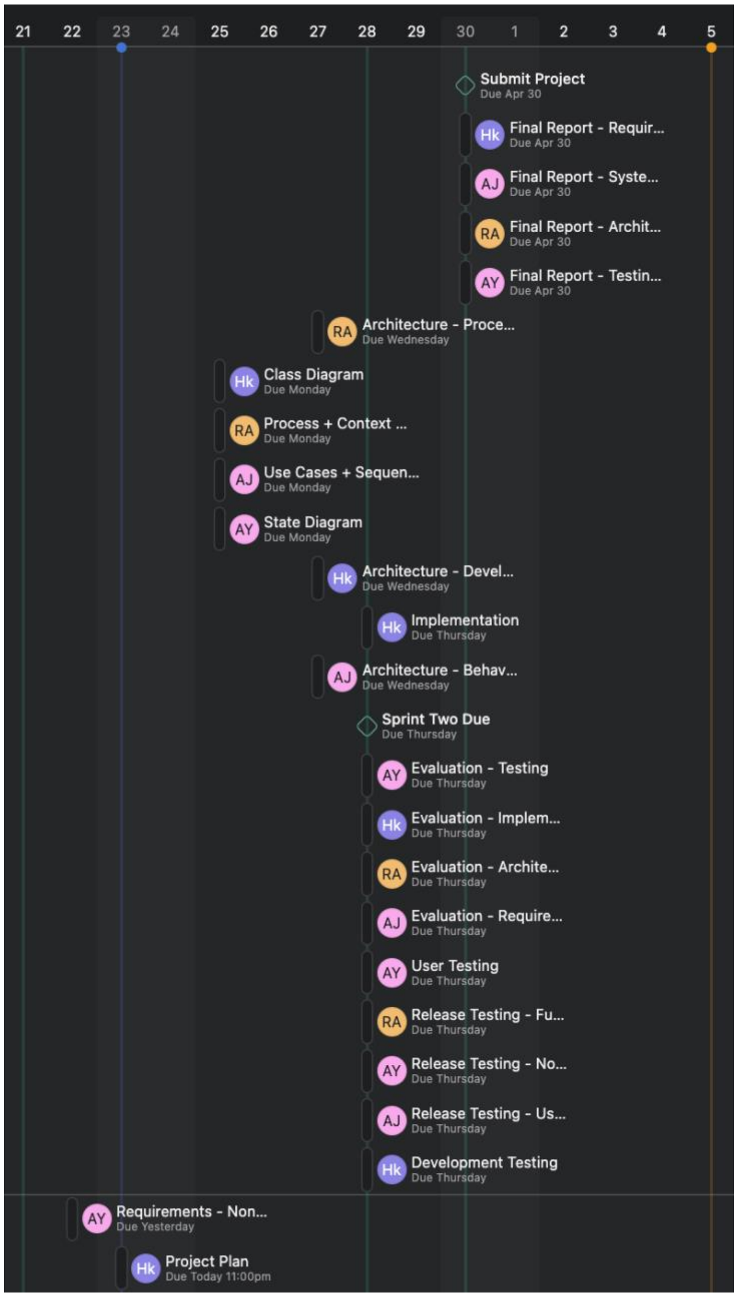
Non-Functional Requirements:

1. Product Requirements:
    - Performance: The system will print a list of illnesses for the user (patient) to select from (API). The system will print an illness diagnosis after the user (patient) inputs their list of symptoms, with an expected time to schedule and print an appoint of 1+(10^-3)n, where n = number of patients in the database. And the expected time for the system to print a doctors schedule will be 1.5+(10^-2)d, where d is the number of doctors in the system
    - Space: The system should be designed to take up less space in regards to storing the list of all the different illnesses, doctors matching the specialty, and storing the appointments in the database. The system uses up to 120 KB as of now but that will increase as more patients/doctors use the system as the database will store more information
    - Dependability/reliability: The system must accept a wide range of valid and invalid inputs for the user's (patient) list of symptoms and still allow for an appointment to be scheduled. The requests on the system are made frequently rather than intermittently, ROCOF is the most appropriate metric to utilize in this case. For example, in a system that processes a significant number of transactions, a ROCOF of less than 10 failures per month might be specified. This indicates you're prepared to accept the fact that an average of 10 transactions per month will fail and have to be canceled.`
    - Security: The system will have a backup process to guarantee that the user can acquire an appointment if the initial appointment process doesn’t work, the list of already booked appointments get erased, or the API call fails
2. Organizational Requirements:
    - Operational: The system will make sure under no conditions that a patient be left unattended for more than 14 days
        - The system should release the appointment/list of additional days/times for the user in a couple of seconds ensuring each patient gets an appointment based on their availability
        - Users may anticipate dealing with a large number of patients in a short period of time, so the system must be designed for maximum efficiency
3. External Requirements:

- Ethical considerations: Without the user's permission, the system may not access personal data
    - The system doesn’t need to access patient medical records to make an appointment unless the patient approves beforehand
    - SSN is not necessary for the user (patient) to enter in order to receive an appointment
    - It is ethical to prioritize certain patients based on age and illness as some may need a quick medical response over others

Project Outline Plan:
The project in total consists of two sprints, where we release initial versions of the system at the culmination of each sprint. For the second sweep of the project (second sprint) we can estimate the time cost and effort of the solution. The project cost will consist of the salaries of all members of the team, along with the continued costs to maintain the system as well as an annual cost to maintain access to components of the system. For this sprint, we introduced a service-oriented architecture where one of the components consisted of a REST API, and for further development, we will begin to introduce REST APIs that allow the user to enter their symptoms and be provided with an illness diagnosis. The maintenance of the system will consist of monthly software updates to resolve any bugs and to create necessary security patches. Lastly, the project will consist of two years of service support, where we will assign some engineers and technical support staff to the customer to ensure all software is functioning correctly and to ensure any technical issues are resolved.

The project is divided into three phases: development, maintenance, and support. The development phase began on April 3rd, 2022, and is expected to last until April 30th, 2022. The development phase will consist of two sprints of the project (and to MVPs), where each sprint will consist of six or so activities (including requirements solicitation, system diagrams and architectures, implementation/testing, and evaluation), each of the activities will be further broken down into tasks assigned to members of the team in coordination with plan-driven development model. The assigned tasks, members the tasks were assigned to, along with the due dates of all tasks are listed below and will be tracked using the project management software Asana.



Submit Project
Due Apr 30

Hk Final Report - Requir...
Due Apr 30

AJ Final Report - Syste...
Due Apr 30

RA Final Report - Archit...
Due Apr 30

AY Final Report - Testin...
Due Apr 30

RA Architecture - Proce...
Due Wednesday

Hk Class Diagram
Due Monday

RA Process + Context ...
Due Monday

AJ Use Cases + Sequen...
Due Monday

AY State Diagram
Due Monday

Hk Architecture - Devel...
Due Wednesday

Hk Implementation
Due Thursday

AJ Architecture - Behav...
Due Wednesday

Sprint Two Due
Due Thursday

AY Evaluation - Testing
Due Thursday

Hk Evaluation - Implem...
Due Thursday

RA Evaluation - Archite...
Due Thursday

AJ Evaluation - Require...
Due Thursday

AY User Testing
Due Thursday

RA Release Testing - Fu...
Due Thursday

AY Release Testing - No...
Due Thursday

AJ Release Testing - Us...
Due Thursday

Hk Development Testing
Due Thursday

AY Requirements - Non...
Due Yesterday

Hk Project Plan
Due Today 11:00pm

The maintenance phase will begin on May 1st, 2022, and last until May 1st, 2023, when the team will be on standby to resolve any defects in the system by releasing quarterly updates to the system. Lastly, the support phase will begin on May 1st, 2022, and last until May 1st, 2024, it will consist of assigning one of the engineers on the development team, along with a technical support specialist to the project to ensure all technical issues that arise over the next two years are resolved. The system will be considered legacy after May 1st, 2023, when a new system can be purchased or more maintenance can be purchased by the customer.

The cost of the project is given below:

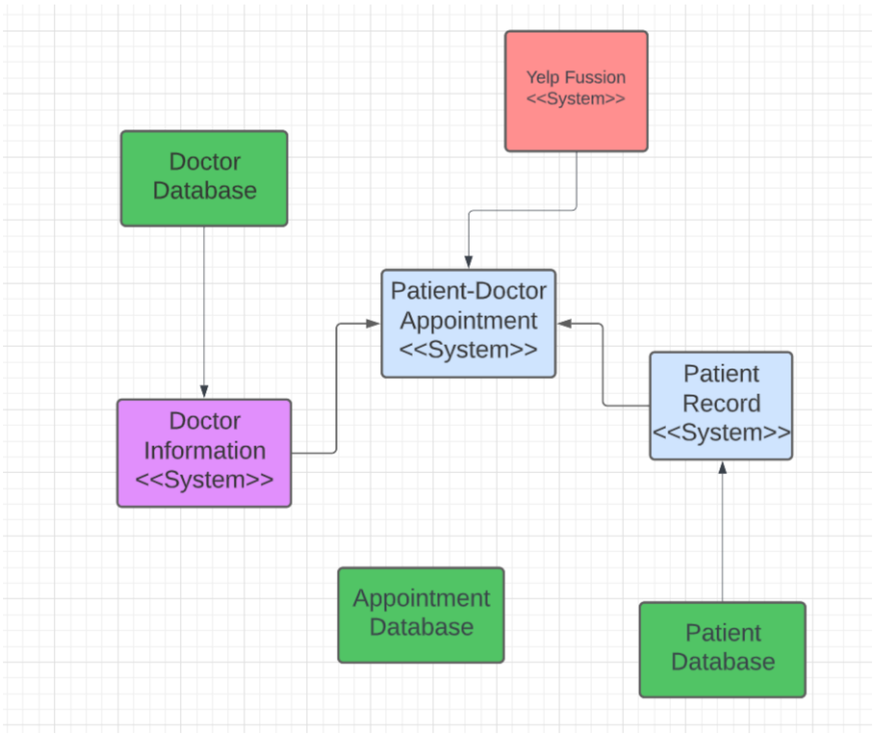
Cost	Rate	Amount/Time	Total
Labor	$300/hr	160 hours	$48,000
Maintenance	$20,000	4 Quarters	$80,000
Technical Support	$40,000	2 Years	$80,000

The labor cost consists of an hourly rate of $70/hr for three engineers, as well as $90/hr for the team lead, where the total hourly cost is $300/hr.

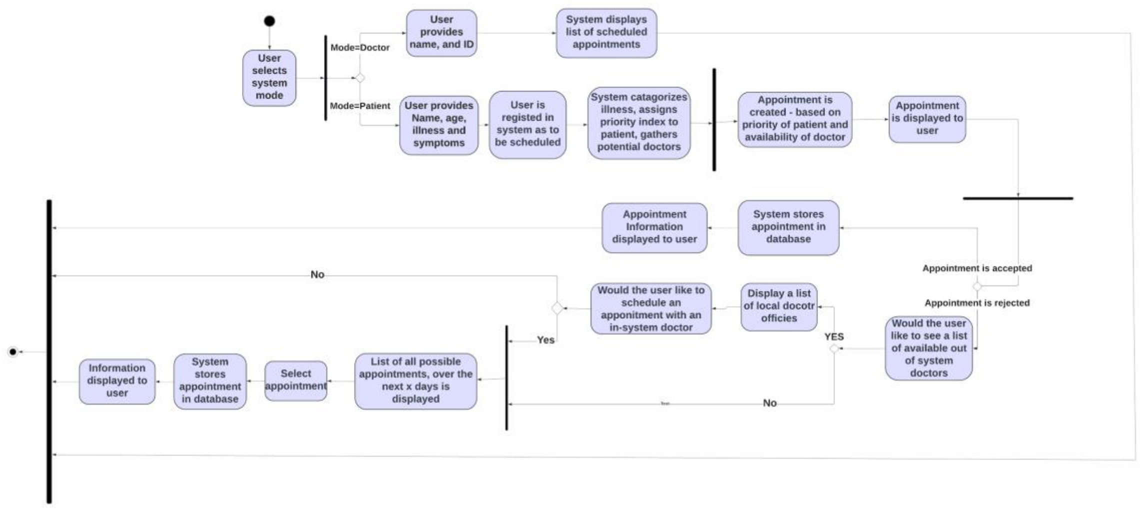
The total expected cost of delivery of this system is $208,000

System Diagrams:

Part A) Context and Process Models


Yelp Fussion <<System>>
Doctor Database
Patient-Doctor Appointment <<System>>
Patient Record <<System>>
Doctor Information <<System>>
Appointment Database
Patient Database

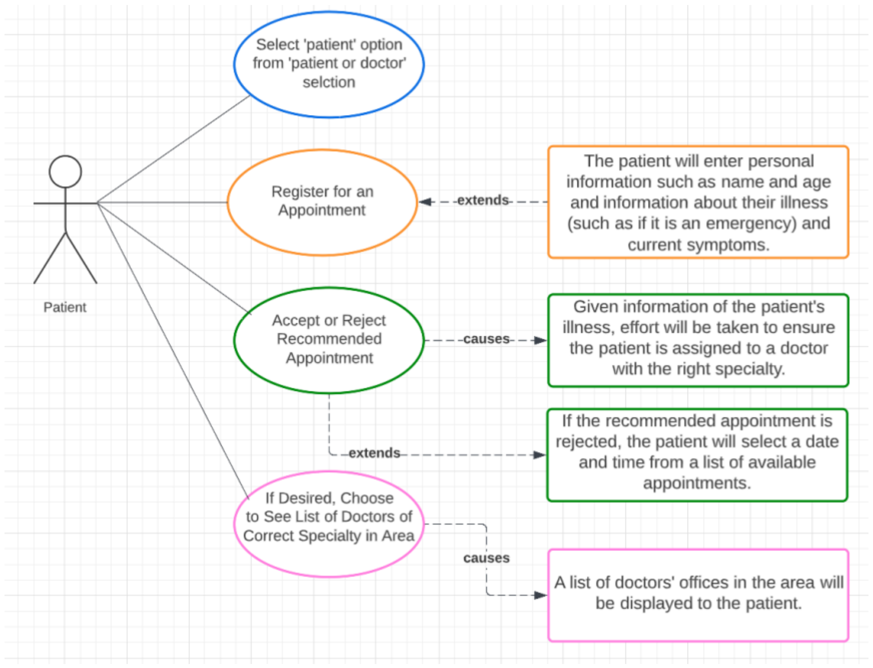
The system shall center around the sub-system which creates an appointment for the patient with the doctor. When the appointment is created, the patient information will be entered into the patient record. Personal information and the specialty of the doctor make up a subsystem as well. Yelp Fussion sub-system will be used to provide the patient with a list of local doctor offices.


User selects system mode
Mode=Doctor
Mode=Patient
User provides name, and ID
System displays list of scheduled appointments
User provides Name, age, illness and symptoms
User is registed in system as to be scheduled
System catagorizes illness, assigns priority index to patient, gathers potential doctors
Appointment is created - based on priority of patient and availability of doctor
Appointment is displayed to user
Appointment Information displayed to user
System stores appointment in database
Appointment is accepted
Appointment is rejected
No
Would the user like to schedule an appointment with an in-system doctor
Display a list of local docotr officies
YES
Yes
Would the user like to see a list of available out of system doctors
Information displayed to user
System stores appointment in database
Select appointment
List of all possible appointments, over the next x days is displayed
No

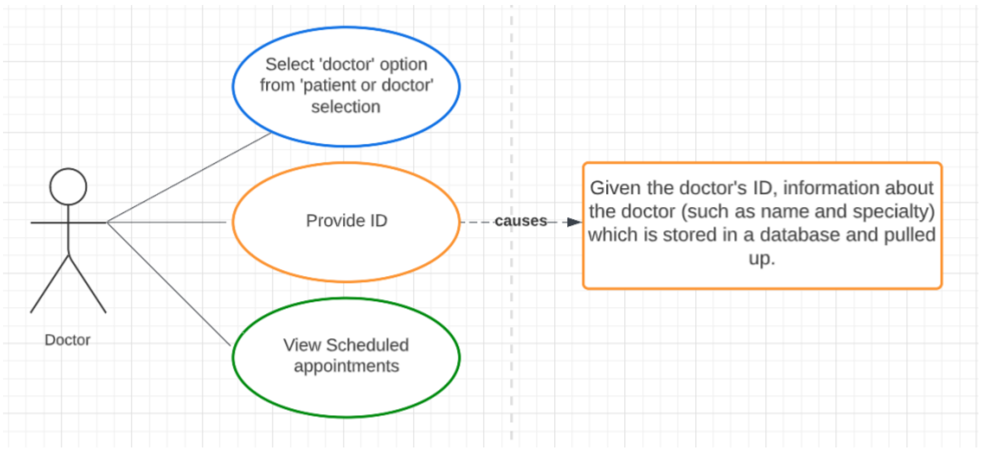
The filled circle indicates the start of the process. At first, the user selects the system mode (doctor or patient). If the selected mode is ‘doctor’, then the user will be asked to enter their name and ID. After that, the system will display a list of scheduled appointments with the doctor.However, if the selected mode is ‘patient’, then the user will be asked to provide their name, symptoms, and illness. Then the user will be registered as ‘to be scheduled’. The system will categorize illness and assign a priority index to the patient and gather potential doctors. Then based on the priority of the patient and the availability of the doctor, the system will create an appointment and display it to the patient. If the user accepts the appointment, the system will store the appointment in the database and display further information to the user. However, if the user rejects the appointment, the system will ask the user if they want to see a list of available out of the system doctors. If the user selects yes, then the system will display a list of local doctor offices and ask the user if they still want to schedule an appointment with an in-system doctor, if the user selects no, then the system will end execution since it only schedules appointments with in-system doctors. If the user selects yes, then the system will display a list of possible appointments to let the user select an appointment. Then the system will store the selected appointment in the database and display further information to the user. If the user after they declined the appointment chose not to view a list of the out-system doctors, then the system will immediately display a list of possible appointments to let the user select an appointment. Then the system will store the selected appointment in the database and display further information to the user.

Part B) Use Cases and Interaction Models:

Use Case Diagrams:

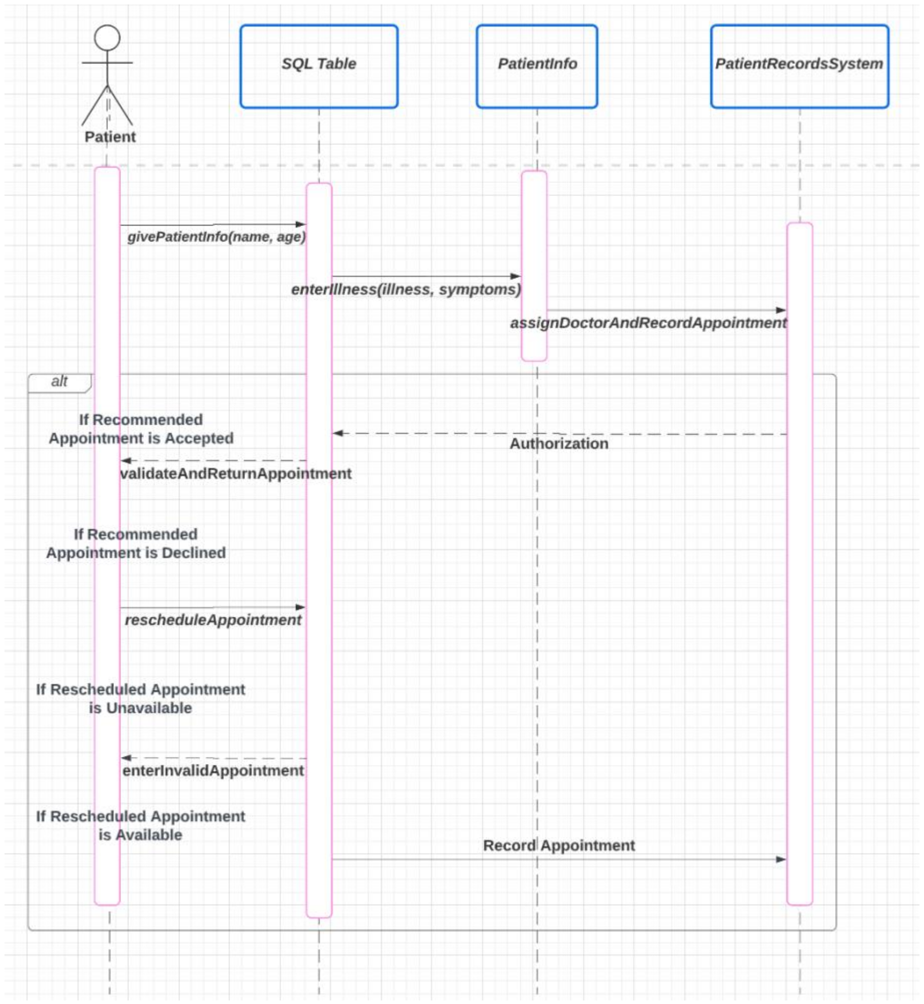

Select 'patient' option from 'patient or doctor' selction
Register for an Appointment
extends
The patient will enter personal information such as name and age and information about their illness (such as if it is an emergency) and current symptoms.
Patient
Accept or Reject Recommended Appointment
causes
Given information of the patient's illness, effort will be taken to ensure the patient is assigned to a doctor with the right specialty.
extends
If the recommended appointment is rejected, the patient will select a date and time from a list of available appointments.
If Desired, Choose to See List of Doctors of Correct Specialty in Area
causes
A list of doctors' offices in the area will be displayed to the patient.

The use case diagram for the patient is updated with the patient given the opportunity to view a list of nearby doctors' offices if desired. This allows for greater dependability and performance.

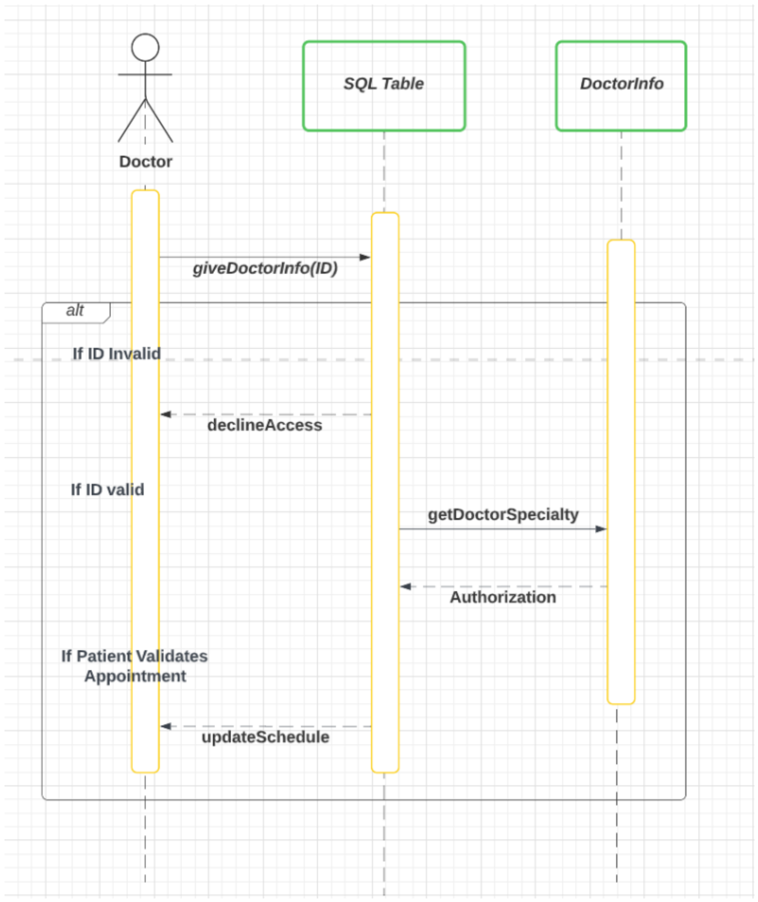

Select 'doctor' option from 'patient or doctor' selection
Provide ID
causes
Given the doctor's ID, information about the doctor (such as name and specialty) which is stored in a database and pulled up.
Doctor
View Scheduled appointments

The use case diagram for the doctor is given some more detail in that when a doctor enters their ID and the system ensures that the ID is valid, the doctor's personal information and specialty are collected from the database

Sequence Diagrams:

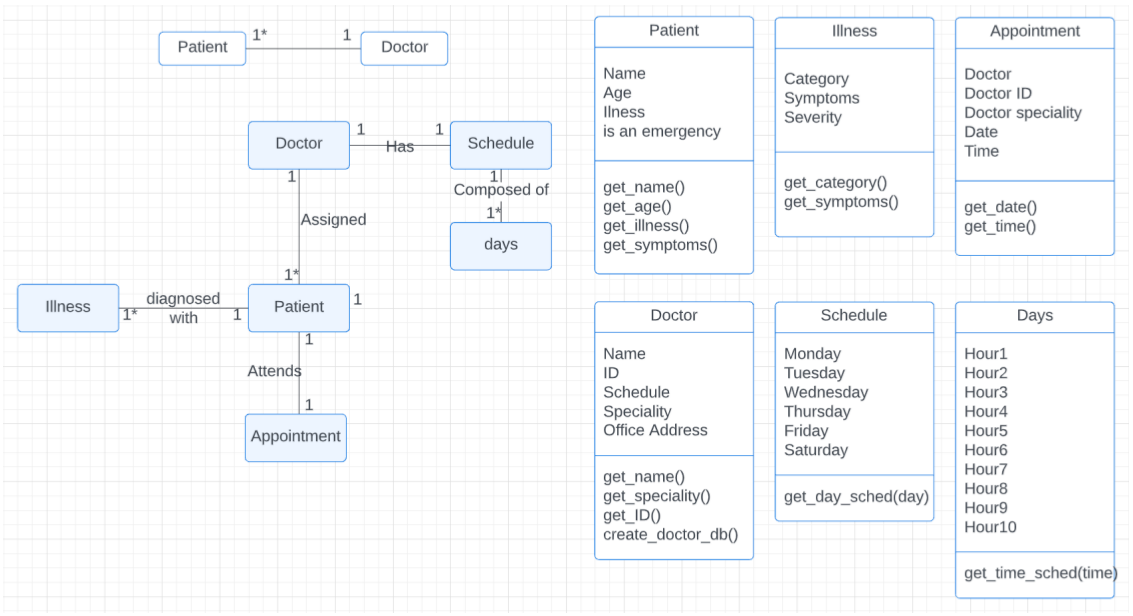

SQL Table
PatientInfo
PatientRecordsSystem
Patient
givePatientInfo(name, age)
enterIllness(illness, symptoms)
assignDoctorAndRecordAppointment
alt
If Recommended Appointment is Accepted
Authorization
validateAndReturnAppointment
If Recommended Appointment is Declined
rescheduleAppointment
If Rescheduled Appointment is Unavailable
enterInvalidAppointment
If Rescheduled Appointment is Available
Record Appointment

The sequence diagram for the patient takes care of what happens if the recommended appointment is declined. The patient then reschedules by entering a date and time given a list of available appointment dates and times. If this entry is invalid (not available date or time), the user is prompted as much and reenters.



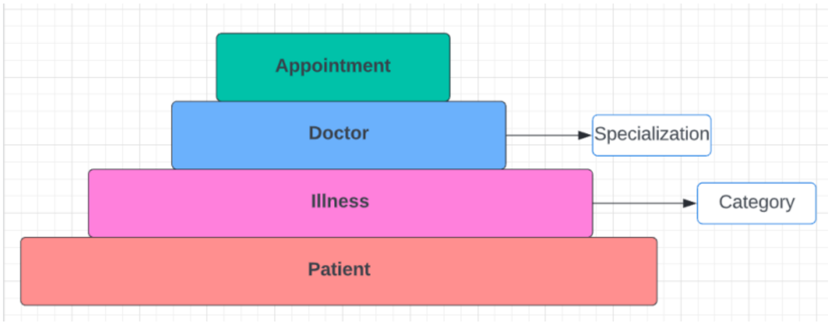
The sequence diagram for the doctor takes into account the ID entered by the doctor. If the ID is not in the database, the user is not given access to the system. If the ID is valid, then the system collects information such as the doctor's name and medical specialty. If a patient accepts an appointment, the schedule of the doctor with the correct medical specialty gets updated with the newly assigned appointment.

Part C) Class Diagram:



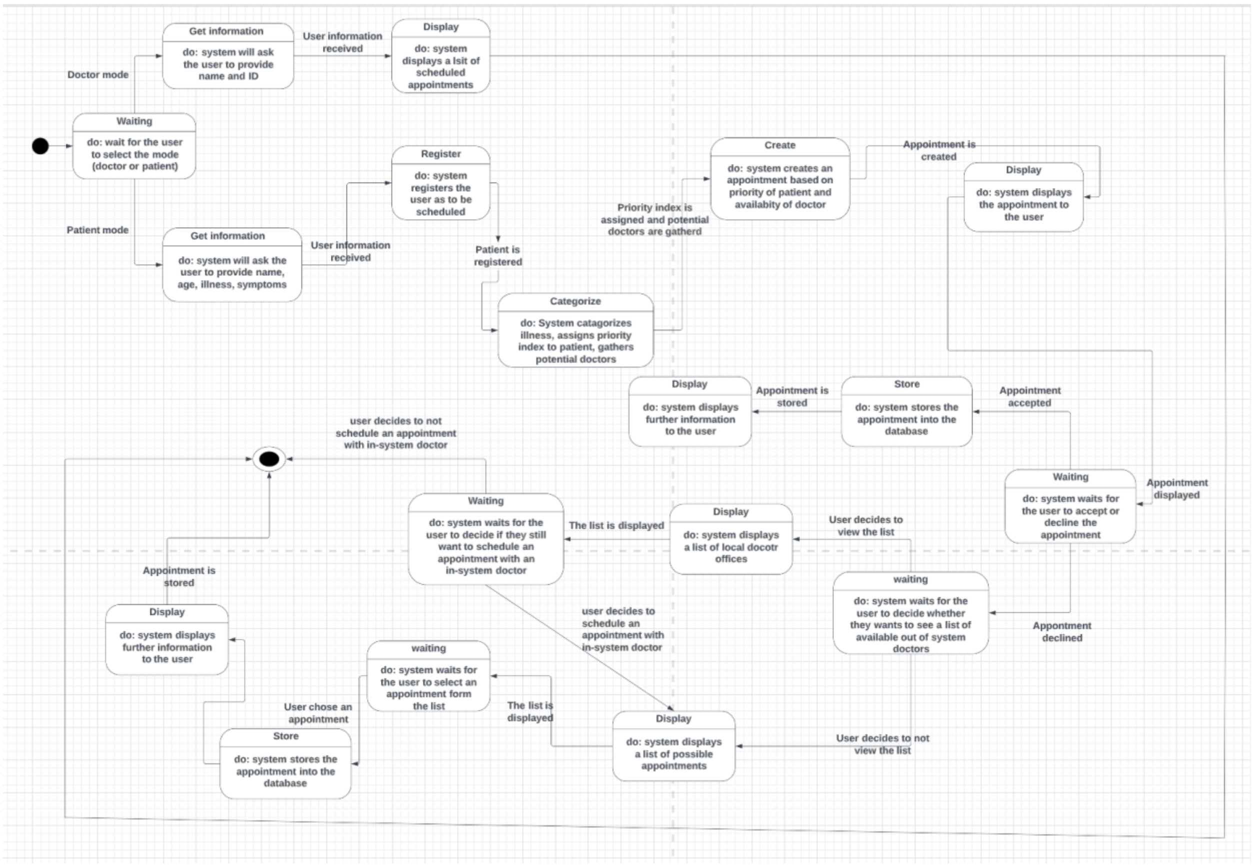
The system shall consist of a series of classes that are linked to each other by certain connections. Each patient class shall be associated with a doctor class, where each patient has one doctor and one doctor has many patients. A Doctor will be assigned to a patient, they will have a schedule (that stores their availability) and each schedule will be composed of day classes (Monday, Tuesday,ect…). The doctor class will also contain relevant attributes, including a name, ID, and office address, and the days class will contain ten attributes representing their availability (True for available, False else) for each of the 10 hours of their workday.

The doctor will also be assigned to a patient, who will be diagnosed with an illness (or multiple illnesses), and each patient will also attend one appointment. The patient class will contain names and age attributes, and the illness class will contain category, symptoms, and severity attributes, while the appointment class will contain doctor name, id, date, and time attributes.



The hierarchy of the class is given above, where the system will begin with a patient (where the attributes are found from the user inputs), and store the illness of the patient (input). Next, the system will jump to the next level class, where it will find the correct doctor for that illness. Internally, however, the system will go from the illness to categorizing it (via an attribute), then it will find the correct doctor-specialty, after which it will find the correct doctor. Lastly, the system will culminate by going to the appointment class where it will create an appointment for the patient based on the specifics entered.

Part D) State Diagram


Get information
do: system will ask the user to provide name and ID
User information received
Display
do: system displays a lsit of scheduled appointments
Doctor mode
Waiting
do: wait for the user to select the mode (doctor or patient)
Register
do: system registers the user as to be scheduled
Patient mode
Get information
do: system will ask the user to provide name, age, illness, symptoms
User information received
Patient is registered
Categorize
do: System catagorizes illness, assigns priority index to patient, gathers potential doctors
Priority index is assigned and potential doctors are gatherd
Create
do: system creates an appointment based on priority of patient and availabity of doctor
Appointment is created
Display
do: system displays the appointment to the user
Display
do: system displays further information to the user
Appointment is stored
Store
do: system stores the appointment into the database
Appointment accepted
Waiting
do: system waits for the user to accept or decline the appointment
Appointment displayed
user decides to not schedule an appointment with in-system doctor
Waiting
do: system waits for the user to decide if they still want to schedule an appointment with an in-system doctor
The list is displayed
Display
do: system displays a list of local docotr offices
User decides to view the list
waiting
do: system waits for the user to decide whether they wants to see a list of available out of system doctors
Appontment declined
Appointment is stored
Display
do: system displays further information to the user
user decides to schedule an appointment with in-system doctor
waiting
do: system waits for the user to select an appointment form the list
User chose an appointment
The list is displayed
Store
do: system stores the appointment into the database
Display
do: system displays a list of possible appointments
User decides to not view the list

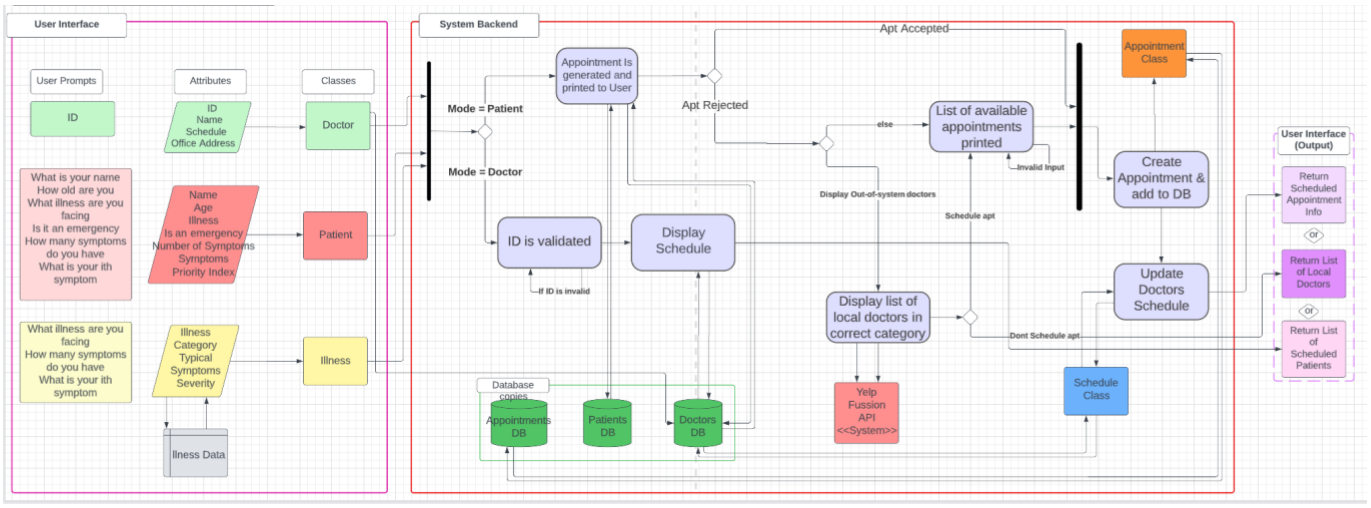
The diagram shows the system states and the events that transition one state to another. The system starts with a waiting state. The system will wait for the user to select the mode (doctor or patient). If the entered mode is doctor, the system will ask the user to provide their name and ID. Entering user information will transition the system into display state, the system will display a list of scheduled appointments to the user. However, if the selected mode is patient, the system will ask the user to provide their name, age, illness, and symptoms. Entering patient information will transition the system into register state, the system will register the user as to be scheduled. Once the patient is registered into the system, the system will categorize illness and assign priority index to patient and gather potential doctors with the right speciality. Assigning the priority index and gathering the potential doctors will transition the system into create state, the system will create an appointment based on the priority index and the availability of the doctor.

Once the appointment is created, the system will transition into display state, the system will display the appointment to the user. After displaying the appointment, the system will transition into waiting state, the system will wait for the user to accept or decline the appointment. If the appointment is accepted, the system will store the appointment into the database. Storing the appointment will transition the system into display state, the system will display further information to the user. If the appointment is declined, the system will transition into waiting state, the system will wait for the user to decide whether they want to view a list of available out of the system doctors. If the user decides to view the list, the system will transition into to display state, the system will display a list of local doctor offices. Once the list is displayed, the system will wait for the user to decide whether they still want to schedule an appointment with an in-system doctor. If the user decides not to schedule an appointment with an in-system doctor, then the system will end execution since it only schedules appointments with in-system doctors. If the user decides to schedule an appointment with an in-system doctor, then the system will transition into display state, the system will display a list of possible appointments.

After the list is displayed, the system will wait for the user to select an appointment from the list. Choosing an appointment from the list will transition the system into store state, the system will store the selected appointment into the database. Once the appointment is stored into the database, the system will display further information to the user. However, if the user decides not to view the list of the available doctors out of the system, then the system will transition into display state, the system will display a list of possible appointments. After the list is displayed, the system will wait for the user to select an appointment from the list. Choosing an appointment from the list will transition the system into store state, the system will store the selected appointment into the database. Once the appointment is stored into the database, the system will display further information to the user.

Architecture:

Part A + B) Architecture - Development and Process View:

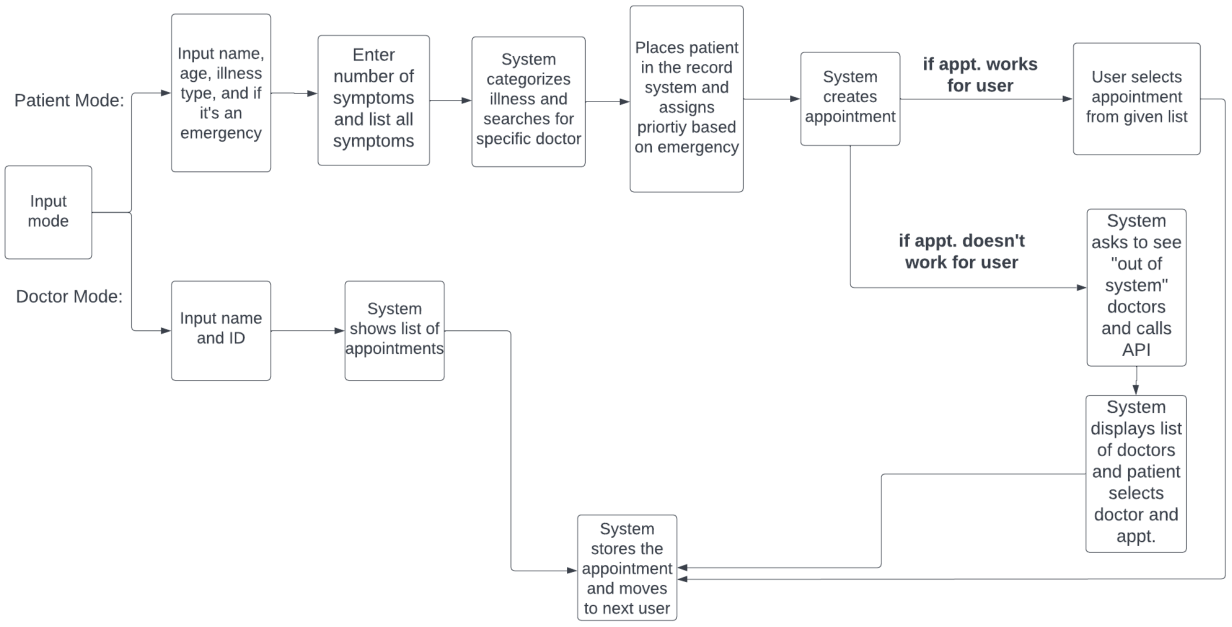

User Interface
User Prompts
Attributes
Classes
ID
ID Name Schedule Office Address
Doctor
What is your name How old are you What illness are you facing Is it an emergency How many symptoms do you have What is your ith symptom
Name Age Illness Is an emergency Number of Symptoms Symptoms Priority Index
Patient
What illness are you facing How many symptoms do you have What is your ith symptom
Illness Category Typical Symptoms Severity
Illness
llness Data
System Backend
Mode = Patient
Mode = Doctor
Appointment Is generated and printed to User
Apt Accepted
Apt Rejected
else
List of available appointments printed
Invalid Input
Appointment Class
Create Appointment & add to DB
ID is validated
If ID is invalid
Display Schedule
Display Out-of-system doctors
Schedule apt
Display list of local doctors in correct category
Dont Schedule apt
Update Doctors Schedule
Database copies
Appointments DB
Patients DB
Doctors DB
Yelp Fussion API <<System>>
Schedule Class
User Interface (Output)
Return Scheduled Appointment Info
or
Return List of Local Doctors
or
Return List of Scheduled Patients

The architecture depicts a more in-depth understanding of the system from an integrated process and development view. The system begins by polling the user with questions (given in the first column), which it then validates (ensures no inputs 'break' the system - dependability) and creates an object of the class given, and then updates the attributes of that class. At this point the UI of the system is complete, and objects have been created that will dictate the running process of the rest of the system. Next, we depict the system using a process view where the system goes into doctor or patient mode. Tracing doctor mode, the user's ID is validated by accessing the values from the doctor database and the system attempts to display the doctor's schedule by accessing the doctor database again (which also holds availability throughout by hour the workweek). At this point the system prints the list (apart from the UI output sub-system) and the program terminates

Setting the mode to the patient, we are then given a recommended appointment which is gathered by accessing the patient object and doctor database (to determine which doctor is available). The user then enters whether they want to select that appointment, if so the system creates the appointment object and stores it into the appointment database, and finally updates the doctor database - where it sets that hour in the doctors' schedule to unavailable. If the user does not select the appointment, they are then asked if they want to see a list of out-of-system doctors (who are doctors of the correct category and local to the area - The system assumes the area is New Brunswick - which is hardcoded). If the user opts to see the list, the system accesses the REST API Yelp Fusion via an API key and some parameters and it sends a GET request to gather relevant data. This data is stored then the system prints to the user (UI Output) and asks (again) if the user wants to schedule an appointment with a doctor (in-system).

If the user opted to not see the list of local doctors (or if they wanted to schedule an appointment with a doctor after viewing the list) then the system will display a list of available appointments by sending data requests to the doctor database, and then the system will allow the user to input a day/time (and validate the input). Next, the system will create the appointment, store it in the database, update the doctor's database, and print it to the user (UI Out).

Part C) Architecture - Behavioral View


Patient Mode:
Doctor Mode:
Input mode
Input name, age, illness type, and if it's an emergency
Enter number of symptoms and list all symptoms
System categorizes illness and searches for specific doctor
Places patient in the record system and assigns priortiy based on emergency
System creates appointment
if appt. works for user
User selects appointment from given list
if appt. doesn't work for user
System asks to see "out of system" doctors and calls API
System displays list of doctors and patient selects doctor and appt.
Input name and ID
System shows list of appointments
System stores the appointment and moves to next user

The behavior diagram depicts the addition of an API that allows the user (patient) to meet with doctors who are not already registered in the database. If the patient is ok with the in-system doctor and the appointment given to them, the system will store the appointment and move on to the next user. If the user would like to see other potential doctors, the system will provide a list of more options, along with their location and phone numbers. The user can then choose a doctor, and a list of available appointments will be printed for them to choose from.

Implementation:

<u>Effort to make the solution dependable</u>

Dependability of a system can be judged by many different properties and end characteristics. some of these properties include the availability of a system, and the reliability of a system, I need the security of a system. H2 is some be considered some of these properties in ensuring it remains dependable. previous releases of the system, it would crash given some invalid inputs and it would overwrite existing appointments inside of the database. as a result, this previous system was not available as often as it should have (the probability that it was up and running was medial), and it was not reliable in that regard. At times the probability it would deliver valid services to the user was low and finally, it was somewhat insecure because existing appointments inside of the database could be overwritten and replaced by new appointments. to solve some of these issues and make the system more dependable overrule, we implemented a set of changes to the code to ensure these properties are met.

To ensure the probability that the system was up and running at any given time was high oh, we validated and checked over all cases that causes the system to exit execution or crash. These cases commonly involved invalid queries to the database, and invalid input data types (integer input when a stream was expected, or vice versa). To solve these we used some specific Exception handling approaches and code in Python. This meant that we had to validate all inputs from the user and all queries to the database, so it says invalid input was detected the system would cycle and pull the user for a valid input until that input was validated oh, and we see this in test cases where we enter an invalid age. if we inspect some of the test cases and the use of the program, we find that when we entered invalid and put the system continues a cycle where it tries to get a valid input from us. If we enter " I don't know when asked for our age, the system will keep asking for our age until it receives a valid input

To ensure that the system is reliable, and the probability that it will deliver the expected Services is high, we implement the above strategy and Valatie all inputs queries and variable calls. if a component calls another component with variable parameters (dependent upon the user's input (then we validate the cool and ensure no exceptions occur and if they do we Implement some error handling code and provide information to the user so they can decide what to do next.

To ensure the system is Dependable we have to make sure it's secure. In this case, that means that old data that's entered into the database cannot be modified without permission from the system oh, that also means that a doctor should not be able to view the schedule of another doctor (so so their ID should be checked an authenticated before they're given access to the database). in the previous system release, the system allowed new users to take an appointment that had already been scheduled with another patient - which represents a security issue. So for this version release, we Implement some checking strategies in the code to ensure that the desired appointment is not scheduled until we validate that it does not already exist inside the database. This means that a user will not be able to schedule an appointment that's already been scheduled for another patient, in the database, and old data is secure. Lost sleep for the doctor use case, we check the ID and put off the doctor and determine if there's a match within the database, at which point if there is, let me provide them access to their schedule and all relevant data. If their ID is not validated or does not exist inside of the database then we request that they re-enter it, and in future releases, we will implement a system termination/lock if an invalid ID is entered x times.

Effort to use an advanced software engineering topic:

To incorporate an advanced software engineering topic, we decided to go with a service-oriented architecture using a restful approach. this man we would incorporate some components in the system that would use the restful approach by accessing via blocks and sending get, retrieve, update, delete requests. we chose to use the rest of API Yelp Fusion, where we would send and get requests with an API key, and some valid parameters. The AP I would return a bunch of data related to a local doctor's offices under the specialization of the doctor required for the category that the illness is categorized in. so if the user decides to view a list of local doctor offices, and their illness is arrhythmia then the system would categorize their illness as a heart condition which would necessitate a cardiologist oh, and the system would retrieve a list of a local cardiologist. For the use of our system, and for Simplicity, we assume that the user is fixed to a location in New Brunswick and all appointments are scheduled with doctors local to the area. we were able to find components that would allow us to automatically retrieve data on local doctors in the area but since we did not have access to schedules and because that approach would result in Louver liability re-elected to hard-code doctors into our system. the system however after providing an appointment recommendation to the user, it will ask the user if they would like to view a list of row system doctors. If the user enters yes the new system will hold the Yelp API with a get request I don't want to pull a day off from the API, for each day a block corresponds to 1 search in their database. The parameters would be a location of New Brunswick, a radius of ten miles, and a keyword search of the category assigned to the user's illness. The API would then return the requested data in the format of a dictionary, and our system would then parse through the dictionary and output relevant to the patient.

Data source:

The system uses a setlist of doctors as well as illnesses, categories, and doctor specializations. As a result, the data source for the doctors was gathered through online searches of doctors local to the New Brunswick, NJ area who belong to the category defined in the dataset. The site ‘healthgrades.com’ was used to gather a list of doctors, as well as all relevant data about them and their practice (including their name, office address, specialty, etc..). All doctors were assumed to be fully available during the days Mon-Sat at the times 8 am-6 pm as is common with many doctors.

The patient's illness was gathered as user input, and the system used a dictionary to map an illness to a category of illnesses and another to then map it onto a doctor's specialization. As a result, some extensive data was required to determine which illnesses were under which categories and which specializations treated which illness categories. We also required some data to develop an algorithm that would determine which patients were of the highest priority, some of this data was generalized (i.e extremities in age necessitate high priority), whereas other parts of this data were gathered directly from sites. Since most of this data is informal and is not really used by doctors or medical professionals (they do not classify an illness as a category and then to a doctor specialization), it was difficult to gather reliable standardized data. As a result, the sources given below are informal but serve the purpose of this project well (As we do not need any formal data as the system won't be used in that manner). The sources are listed below.

https://clinicaltables.nlm.nih.gov/demo.html?db=conditions

https://www.altexsoft.com/blog/healthcare-api-overview/

http://www.health2api.com/?p=9102

https://almostadoctor.co.uk/browse/surgery

Testing
<u>Part A) Development Testing - Automated tests</u>

We break down development testing into three groups, unit tests, component tests, and system tests. The tests are executed in the python file named ‘Test_File.py’. All tests were programmed into the file (and automated), and the documentation of all tests, along with the validation of the result is given in detail here.

1. Unit Tests

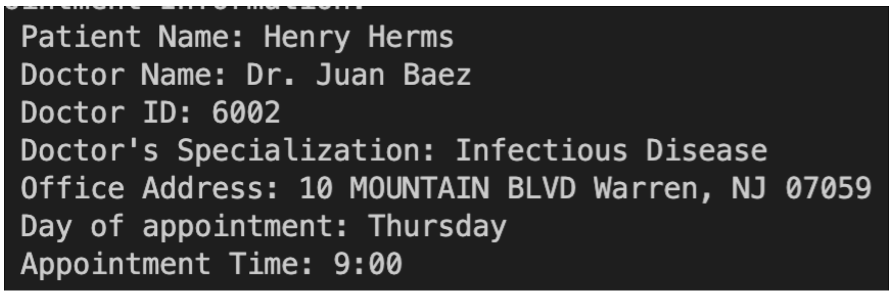
   For these sets of tests, we use the partition testing method, partitioning the system into multiple groups, where each group has a set of common inputs. We then create multiple tests for each of these groups. After executing the tests, we can begin to validate the result and determine whether the system response matches what was expected. We can partition the system inputs into four groups: mode of use, illness, whether its an emergency, symptoms entered, and whether the suggested appointment was selected

   Since we will focus on particular inputs/groups of the system, the name of the patient will be arbitrary.

   <u>Test One:</u>

   Inputs: mode = “Patient”,name = “”,age =30, illness = “Tetanus”, is_emergency = False, symptoms = “”, apointment_accepted = True

   The illness is classified as an infectious disease, and referred to an ID doctor, the system sends a request for an ID doctor to the database and then uses some logic to determine when to schedule the appointment. The output is given below.



```
Patient Name: Henry Herms
Doctor Name: Dr. Juan Baez
Doctor ID: 6002
Doctor's Specialization: Infectious Disease
Office Address: 10 MOUNTAIN BLVD Warren, NJ 07059
Day of appointment: Thursday
Appointment Time: 9:00
```

For the first test, we focus on the mode of the user group, where we test the system with the mode set to the patient. We create a patient object and call a sequence of functions to observe the output. In this case, we will accept the output appointment.

Test Two:

Inputs: mode = Doctor, ID = 6001, apointment_accepted = True

For this test, we focus on testing the model inputs and seeing if the system responds as expected when the mode is set to doctor. In this case, the system will use the ID supplied in the test (6001) and select the doctor column where id=6001 through a SQL query to request the DB.

The system outputs a list of 16 patients, along with the time and day of their appointment. This test can be considered passed, as the system provided a list of patients scheduled, which constantly updates when a new patient uses the system

Test Three:

Inputs: mode = “Patient”,name = “”,age =30, illness = “Diabetes”,is_emergency = False, symptoms = “”, apointment_accepted = True

For this test, we focus on the illness, whereby we use a moderate chronic illness, with no symptoms and a low-risk age and observe the system response. In this case, we are provided an appointment with an Endocrinologist (doctor belonging to the endocrinology category) in New Brunswick on Friday at 2:00 PM (14:00). We can safely say this unit test has passed as the system classified the patient as low risk and gave them an appointment later in the week.

Test Four:

Inputs: mode = “Patient”,name = “”,age =22, illness = “OCD”,is_emergency = False, symptoms = “”, apointment_accepted = True

For this test we provide an extremely low-risk patient, with OCD as their illness and an age of 22 (considered low risk), expecting the system to provide an appointment on Friday or Saturday. The system responds by scheduling an appointment with Dr.Kenneth Burns, in Highland Park, for Saturday at 9 am. We can validate the test, as the behavior of the system in this test matches our expectations.

Test Five:

Inputs: mode = “Patient”,name = “”,age =40, illness = “Coronary artery disease”,is_emergency = False, symptoms = “”, apointment_accepted = True

In this test, we increase the age and provide a high-risk illness to determine if the system will prioritize the patient for an early appointment. In this case, the system provides an appointment with a cardiologist for Tuesday at 3:00 pm. We can say that this unit test has passed as the provided appointment was with the correct doctor, at a time early in the week.

Test Six:

Inputs: mode = “Patient”,name = “”,age =30, illness = “Celiac disease”,is_emergency = True, symptoms = “”, apointment_accepted = True

For our next case, we vary the is_emergency value (bool) to true and observe the behavior of the system. In this case, we provide a medium-low risk disease and a low-risk age, and the system responds by creating an appointment with a Gastro doctor, for Wednesday at 14:00. Although the test can be considered to have passed, since the user deliberated their condition as an emergency it can have been moved to either Tuesday morning or Monday evening. We also note that the address of the doctor is not provided in this case, which is abnormal.

Test Seven:

Inputs: mode = “Patient”,name = “”,age =30, illness = “Celiac disease”,is_emergency = False, symptoms = “”, apointment_accepted = True

For this test, we focus on comparing the results of this test and the previous one. Since this test is identical to test six, with the difference being the state of the variable is_emergency, we can validate the behavior of the system in this case if it matches the previous test but with an appointment scheduled later in the day. The appointment provided is on Thursday at 2:00 pm, which matches our expectations, but confirms that test six provided an appointment too late in the week.

Test Eight:

Inputs: mode = “Patient”,name = “”,age =30, illness = “Celiac disease”,is_emergency = False, symptoms = “weight loss”, apointment_accepted = True

For this case, we modify test seven by adding a symptom (weight loss), which is alarming for celiac disease. Observing the output, we find that the system provides an appointment for Wednesday at 11:00 am which validates our expectations, showing that the system is able to vary the priority of appointments based on not only age and illness but also symptoms.

Test Nine:

Inputs: mode = “Patient”,name = “”,age =30, illness = “Celiac disease”,is_emergency = False, symptoms = [“weight loss”,”severe abdominal pain”], apointment_accepted = False

Finally, for this test, we focus on the case where the user does not accept the provided appointment, and we observe the behavior of the system. In this case, the system provides a list of available appointments for that particular doctor. We can validate the output by checking that the patient is able to successfully choose their own appointment (and be registered in the system) and if the outputted list of appointments reflects the changes from previous use of the system. Checking for these two conditions, we find them both to be true, leading us to the conclusion that the system has passed the final unit test

2. Component Tests

The component tests will focus on providing invalid or incorrectly formatted inputs to the SQL tables stored in the databases. It will also focus on testing procedural interfaces which are being encapsulated and packaged for use by later interfaces in the system. For the doctor-patient appointment system that would be the functions used to create and store patient and appointment objects.

Test One:

For this test, we focus on the methods dealing with creating and assigning priority values to patient objects. We Design a test and create a patient object with the following inputs

Inputs: name=(arbitrary), age = “I don't remember”, illness = “Schizophrenia”, is_emergency=False, “No I'm tired”

The system is run manually, through the user_inputs() method and it operates successfully. When the age is entered, it prints a message and requests a valid input (cycling until the input is valid), and the illness and symptoms are accepted as is.

Test Two:

For this test, we focus on the illness and symptoms inputs, where we provide an illness not stored in the dictionary by the system.

Inputs:name=(arbitrary), age = “25”, illness = “Foot disorder”, False,”I'm tired”

For this test, the system operates normally and passes the test. The illness is not listed inside of the illness dictionary, as a result, the system classifies it as a general illness that belongs to a doctor of the general practitioner category. The outputted appointment fits these criteria.

Test Three:

For this test, we focus on testing the appointment creation components which create appointment objects and store them in the database. Since the appointment objects are dependent on the patient objects, we can test the appointment components by updating attributes stored in instances of the appointment class. Providing an invalid ID input will allow us to test the ID authentication component for validity

Inputs: Patient object(name,25,” Diabetes”, False,””), doctor ID=4830,

For this test, we alter the doctor id to one that is not contained in the database. In this case, since the doctor request component is supposed to validate the ID before pulling data from the database, the system encounters an issue in this test. The entered ID is invalid, and the system detects it, then prints the message to the user and polls them until a valid ID is requested. The test has passed, however, for security purposes, the system should exit or lock if the user enters three invalid IDs

Test Four:

Input: doctor_ID = 6001
For this test, we focus on the doctor use case and subsequent components, as a result, we attempt to access the schedule of a doctor with a valid ID. When running the test, the system prints a schedule for the specified doctor with the corresponding ID. Upon inspecting the scheduled appointments we find them to be in line with the previous tests, where those patients scheduled appointments with this doctor (Dr.Ronald Nahss). The system has passed this test

Test Five:

We can focus on the date and time variables by supplying incorrectly formatted inputs, or alternative ways of entering the date and time

Inputs: date = 04/25/22, time = 12:30 pm

For this test, we follow a similar protocol as the previous component tests, where we call the user_inputs() method and enter inputs manually. In this case, the date is invalid and the system prints an error message, and requests a valid date input - and it cycles so long as the input entered is invalid. Then it moves on the time, where it accepts the time input and schedules the appointment. The system only schedules appointments at hour intervals, so the requested time is rounded down, and the appointment is scheduled for 12pm. The system has passed this test, although it should have asked the user if they would be okay with an appointment time of 12pm (before scheduling it).

Test Six:

Inputs: Random name, age=30, illness = Arrhythmia, no emergency, no symptoms, a recommended appointment is rejected, local doctor offices list requested

In this test, we provide some arbitrary patient parameters, and more importantly focus on testing the API interface components, and the logic behind it. When asked by the system, we reject the appointment, and then we enter 'yes' when asked if we would like to view a list of local doctor's offices. When running the test, the system behaves as expected, and finishes by printing a list of doctors local to New Brunswick, where they are all (for the most part) cardiologists, that treat the inputted illness. Normally the system would ask if we would like to continue scheduling our previous appointment, and if we enter no then the system would terminate. In this test, we want to enter "no" as our input, but we do not run that code in the test, since this results in the whole testing process terminating.

Test Seven:

Inputs: patient info=same as the previous test reject the appointment, schedule appointment

This test is identical to the previous test, with the main difference being that we request the system schedules us an appointment (after providing a list of local doctors offices/doctors). Here, we want to test whether the system can return control back to the main branch (where it's as if the API was never accessed). When running this test (manually), we are given a list of doctors, similar to test six, and then we are provided with a list of available appointments for a cardiologist in the system. We then select an appointment (Valid inputs) and we are scheduled in the database, and all information is printed. The system passes this test and all unit + component tests with flying colors.

3. System Tests

The system implemented during this sprint focused on in-house components and sub-systems, developed through an engineering in-house process. As the system was developed individually the systems tests for this system will be few. We note that the biggest components that were 'tacked' onto the system are the components that allow a patient to reschedule their appointment if they do not like the one provided by the system. As a result, we can define some tests to determine how the system behaves given some parameters relating to displaying available appointments and creating a new one.

<u>Test One:</u>

Since the system provides a method to collect the requested appointment date and time from the user, we can alter the input parameters and test whether that affects the system altogether - specifically when it queries the table for doctor schedules. We ignore the specifics of the patient object and use a random object with age=30, illness = “Diabetes” and is_emergency=False.

Input: date = “None”, time = “5pm”

For this test, we execute the user_inputs() method, and enter inputs manually. When entering the given date, the system prints a statement and asks for valid input. This cycle continues until a valid input is given.

<u>Test Two:</u>

We can provide the system with another test to provide a valid date parameter and an invalid time parameter to query the table with.

Input: date = “Monday”, time = “None”

For this test, we find that the system behaves in an almost identical manner to the previous test. The results of this test are also in line with our initial expectations. The system prints an ‘error message’, letting us know that the inputted time is invalid and we must enter a valid time. This cycle continues so long as our input is invalid

<u>Test Three:</u>

For this test, we create a case where the inputted date and time are both valid and formatted correctly but they are known to be already occupied by another patient. In this case, we will provide date/time inputs that are not available, and observe the system's behavior. We can do this by replicating the test from unit test nine, and checking whether the system lets us know that the date/time is not available.

Inputs:age = 60, illness = “celiac disease”, False, [“weight loss”,”severe abdominal pain”], date = Monday, time = 10am

This test can be considered one of the most important among all development testing, as it checks to see whether the system is able to ensure appointments that are scheduled with doctors are not overwritten or changed. The system must ensure that if a user is given an appointment with a particular doctor, at a particular date/time, that a user, later on, cannot select that appointment - then two patients will be given the same appointment at the same time - causing a major issue. This system passes this test, unlike in the previous mvp, and it returns a message to the user letting them know that the requested appointment is invalid/unavailable. And it requests new inputs, and cycles until they are validated.

Test Four:

We can create a system test, by providing valid inputs identical to those of the previous test, to check whether the system doesn't allow a user to schedule an appointment that has already been scheduled by another patient. This test is generic and just schedules the appointment into the database

Inputs: date = Friday, time = 10 am

For this system test, we use the same methodology of the previous test, but now we try to create a new appointment at a specific date/time. We expect the appointment to be entered properly, but the main test occurs in test five, where we are checking to see if an appointment that has been entered into the DB, is reflected immediately after being entered. This test passes, and our appointment is scheduled in the DB.

Test Five:

Inputs: date = Friday, Time = 10 am

For this system test, we repeat the previous test, this time checking to make sure that duplicate appointments cannot be created, and that previously scheduled appointments cannot be scheduled by a different patient (once an appointment is set - it must be fixed - unable to be overwritten by another patient). When we run these tests against the system, we are given a prompt that tells us the date/time inputs we entered are invalid - where we must re-enter new date/times. The system has passed the test, meeting one of its main functional requirements. This case is similar to a user scheduling an appointment (and their data being uploaded to the DB), then another user attempting to schedule an appointment for the same time. What we are really testing is whether the system reflects changes in the DB once an appointment object is created, and it does (so the system passes this test once again)

Test Six:

Inputs: name = “Cardiology”

For this last system test, we test the REST component that calls the Fusion API to GET valid data. We modify one of the inputs for this system (the name to search in the API), where we enter a valid and acceptable input. This input would be created by the system, where it is copied from the ‘category’ variable (assigned to the patient's illness). When running the test, the system passes and no issues are detected.

Part B) Release Testing:

<u>System Tests:</u>

<u>Test One:</u>

We design a test to determine if entering different illness inputs for a patient will result in being provided with a doctor belonging to a different specialty. We know that if a doctor with a different specialty is outputted, then the system is able to meet the requirement of categorizing illnesses by N categories and then assigning them to doctors. Based on the results of unit tests one and three, we can see that for different illnesses, different doctors are assigned to appointments and the system is able to categorize illnesses into N categories, fulfilling the requirement.

<u>Test Two:</u>

We design a test that will allow determining if the illness is categorized into the correct (or relevant category). To do so we observe unit tests one and three again, where the inputted illnesses are distinct. In this case, the system provides a doctor of a distinct category for each patient, where the doctor's specialty correlates very closely to the inputted illness. I.e for diabetes illness the system provides an endocrinologist, and for Arrhythmia the system provides a cardiologist. The system has passed this requirement test.

<u>Test Three:</u>

For this test, we focus on checking to see if the system is able to catch distinct illnesses and then assign categories of illnesses to doctor specializations. In order to test this requirement, we can observe some of the systems tests detailed above. for the system test with test cases that use different illnesses we find that the outputted doctor specialty is distinct in both cases. Similar to the previous System test we know that if we vary the illness in front then we should expect to find the specialization of the doctor output it to be distinct. if this is the case then we know the system has internally assigned the illness to a category and then used a database to find the correct doctor of the correct specialization that deals with that category of illnesses. Based upon the outputs of the system test we know that this requirement has been fulfilled by the system

<u>Test Four:</u>

For this requirement test, we focus on determining if the system is able to prioritize patients based on their inputs. This means that if a patient is of a high-risk age, has high-risk symptoms, or has a high-risk illness (or if they enter an emergency input) then the system will be able to prioritize them and assign them to an appointment earlier or later depending on the specifics of their case. We can observe component test four, where the is_emergency input is true and the patient is of high-risk age, illness and symptom. We use this test case to test this system requirement, noting that the outputted appointment is early in the week meaning that this requirement has been fulfilled by the system.

Test Five:
To ensure that the system is dependable and no inputs will cause the system to fail, as stated in the requirement, we can observe the output of some of the later system tests. looking at the later System test, where we enter in manual inputs we find that when we enter in invalid input, or out of bounds inputs, then the system is able to enter a loop where they continue to pull the user until valid inputs are entered and checked. And it case we can summarize and say that the system has passed this test and the requirement has been fulfilled.

Test Six:
For the last system requirement test we can focus on whether the system is able to incorporate a service-oriented architecture, to determine if this has been fulfilled, and in order to avoid having to sift through code, we can test the calls to the REST API and ensure that they are fulfilled and processed correctly. One of the system tests is able to cast this requirement, but only by accessing the access API method are parameters paused to the rest API. Inspecting this test we find that the system is able to deposit, which means that the system has fulfilled this requirement and does incorporate a service-oriented architecture to satisfaction

Functional & Non-Functional Requirements Tests:

Testing Requirement 1:
Is the user able to enter illness information into the system and get an appointment?

According to unit test number 1, the user is able to input their name, age, illness, and whether they have an emergency or not and the user gets appointment information (Doctor name, Doctor ID, Doctor’s specialization, Office address, Day of appointment and Appointment time). This requirement has been satisfied.

Testing Requirement 2:
Does the system allow the user (doctor) to enter their name and ID and get a schedule of their appointments with their patients?

According to unit test number 2, the user is able to input their name and ID and view a list of patients along with the time and day of their appointment. The list constantly updates when a new patient uses the system as well. Therefore, this requirement has been satisfied.

Testing Requirement 3:
Is the system able to create an appointment for every patient with a doctor with the right speciality?

According to unit test number 1, the user is able to enter their illness and get an appointment with a doctor with the right specialty. The user entered their illness tetanus and the system classified it as an infectious disease and created an appointment with a doctor whose specialty is an infectious disease. This requirement is satisfied.

Testing Requirement 4:
Does the system ensure that no patient must wait more than 14 days to get an appointment?

The system assigns to each patient a priority index depending on their illness, age, and symptoms. This index determines who is scheduled first and who is scheduled last. This guarantees that patients are seen in the proper order and as soon as possible and that no patient waits for more than 6-7 days.

Testing Requirement 5:
Does the system allow patients to reschedule appointments?

Once the appointment is presented, the user has an opportunity to decline the suggested appointment and pick from a list on his/her own. This ensures that the patient has a choice of when they would like to get the appointment if the suggested one does not seem fit to their liking

Testing Requirement 6:
Does the system ensure that each patient gets a unique appointment?

According to systems test number five, it is evident that when a patient selects a date and time that is already entered in the database to be scheduled for another patient, the system outputs an error message letting the user know that the input is invalid and that they must select a different day and time for their appointment. This ensures that each patient gets a separate appointment and that no two patients are scheduled at the same time.

Testing Requirement 7:
Does the system still function when invalid inputs are entered?

According to component test number three, it is clear that when the user (doctor) enters an invalid ID, the system prints an error message and cycles through until a valid ID number is entered. After a certain try, the system will terminate. According to component tests number two, when an illness that is not stored in the database is entered, the system still allows for the patient to receive an appointment but instead of having a specific doctor, it schedules it with a general practitioner. This still ensures that the system functions correctly even when invalid inputs are entered by different users.

Testing Requirements 8:
Does the system check for wrong format inputs before accessing the database?

According to component test number 5, when the user enters invalid date the system will print an error message, and request a valid date input, and it cycles as long as the input entered is invalid. Then it moves on the time, where it accepts the time input and schedules the appointment. This requirement is satisfied.

Testing Requirement 9:
Are patients prioritized based on the risk of their illness, age, and/or symptoms?

According to unit test number three, it is clear that when the inputs show a low-risk patient, their expected appointment is scheduled later in the week. Similarly, if the user classifies as a high-risk patient, their appointment gets scheduled earlier in the week ensuring that the system prioritizes the patients accordingly.

Testing Requirement 10:

Does the system allow the user to view a list of out the system doctors?

According to component test number 10, In this test, When the user rejects the appointment, and then they enter ‘yes’ when asked if they would like to view a list of local doctor's offices. The system will print a list of local doctors in their area. This requirement is satisfied.

Part C) User Tests:

All of the unit test cases were passed by the system. According to the unit tests, patients are scheduled based on the severity of their illness's risk factor. Low-risk patients are scheduled later in the week, whereas high-risk patients, as well as their age and symptom list, are given higher priority when it comes to arranging appointments. The tests also show that the correct doctor is allocated to the type of disease that the user has specified. The system also ensures that a patient may quickly reject the first appointment selection and choose a new day and time that suits them better.

The system correctly accepts invalid inputs for component tests and still allows the user to schedule an appointment. If the user does not enter a valid age, the system will display an error message until the user enters one. If the illness isn't stored in the system, it will assign the user to a general practitioner while still generating a list of appointments to be made. If a doctor's ID is incorrect, the system will display an error notice and then lock after a specified amount of invalid entries.

During system testing, patients were able to successfully reschedule their appointments if necessary. These tests also confirm that if a slot is already taken, the system informs the user that the time and date they selected are incorrect and that they must choose another time and date.

Nonetheless, the system is still functional, and it can create appointments to patients. A doctor can also obtain a schedule of their various patient appointments. I believe the system should be released as an initial version. The difficulties from sprint 1 have been resolved, and no two patients have the same appointment. Any invalid inputs, such as illness, symptoms, or doctor ID, do not prevent the system from working.

Evaluation:

Requirements Engineering

- Are all requirements listed clearly?

All requirements are listed clearly and help to make the system design more lucid and specific. The requirements serve as a guide for the software implementation.

- Easy to understand and written thoroughly?

The requirements are written completely. All requirements covered under the project problem are laid out and thoroughly defined and expanded upon.

- Are the requirements and the basic functionality of the system similar?

Yes, the basic functionality covers all of the requirements, fulfilling the main purpose of the system.

System Modeling

- Are there enough diagrams for each part of the system?

Yes, there are enough diagrams for each design and architecture section.

- Each diagram shows a clear understanding of the different parts of the system and how it will operate.

All diagrams are clear and help to simplify the functionality of the software. Analysis of the diagrams show that they are consistent with the code implementation and the purpose of the system.

Architectural Design

- Are architectural designs available?
  The architectural designs are available. The sprint includes a development and process view diagram and a behavioral view diagram. The diagrams show how the software will be organized. There is an explanation below each diagram to explain these diagrams and show how the system will work.

- Designs are well organized and portray a clear understanding of how the system will work
  Each diagram is well organized and provides a clear understanding of how the system will work. For example, the architecture and process model provides a clear understanding of how the system will work. The system begins by asking the user to answer the questions given in the first column. The system will validate these inputs and create an object of the class given, and then update the attributes of that class. Then the system goes into doctor or patient mode. Tracing doctor mode, the system will use the doctor database to validate the user's ID and then the system will display the doctor's schedule by accessing the doctor database again. After that the program will terminate. Setting the mode to the patient, the system will recommend an appointment which is gathered by accessing the patient object and doctor database. Then the user enters whether they want to select that appointment, if so the system creates the appointment object and stores it into the appointment database, and updates the doctor database. If the user does not select the appointment, the system will ask them if they want to see a list of out-of-system doctors. If the user chooses to see the list, the system accesses the REST API Yelp Fusion to display a list of local doctors in their area. Then the system will ask the user if they still want to schedule an appointment with an in-system doctor. If the user chooses to not see the list of local doctors (or if they wants to schedule an appointment with a doctor after viewing the list) then the system will display a list of available appointments by sending data requests to the doctor database, and then the system will allow the user to input a day/time (and validate the input). Next, the system will create the appointment, store it in the database, update the doctor's database, and print it to the user.

Design and Implementation

- Code is clear and easy to understand?

The code is clear and easy to understand, it is organized in a way that makes it easy to follow, both in class hierarchy and structure, as well as the organization of code into multiple files. The logic used in the code is also somewhat simple, and not too difficult to follow and there is little redundancy in the source code making it easier to read. Comments are added throughout the code, and everything follows a standard format to provide explanations to other developers

- Code works with no errors

The code works well, with no errors. The only issues that come up are when the user has not installed the correct libraries onto their machine, if they have not downloaded all files into the same directory, or if they do not run the doctor.py file before the driver. However, the code works with no errors, where the only errors that occur are the result of incorrect usage of the system as a whole. No noticeable bugs or defects are present in the code, it was developed using a test driven development model, to ensure every line works as intended and wont 'break' under certain circumstances.

- Variables and functions are used correctly.

All variables are and functions are used correctly, everything is organized and structured well. All variables and functions are also named according to their intended purpose and what they contribute to the system as a whole. As a result, the source code is easy to follow and understand, with some complexity arising due to the strenuous nature of the algorithms/logic employed. Get and set methods are also used to store and update data, and in other instances (to make code simple), attributes are accessed through the self attribute. Each function corresponds to a set of tasks, where some focus on gathering inputs and validating them, others focus on interacting with different components, and others focus on the main tasks achieved by the 'Backend'

Software Testing

- Outputs as expected.

The system produces the expected results. The patient can enter information about their illness and be scheduled for an appointment. They also have the opportunity to select a different day and time if needed. The specialized doctor is assigned and an appropriate date for the appointment is set based on the patient's category and priority of illness, age, symptoms, and if it's an emergency.

- Multiple test cases are used

Yes, approximately 25 test scenarios are employed in total. Unit tests, component tests, and system tests are all included. The numerous tests demonstrate a wide range of both valid and invalid inputs. The results are displayed to demonstrate that the system is still capable of running smoothly and efficiently.