Sprint One:

Requirements Engineering:

Part A) User and System Requirements:

User Requirements:

The system will schedule appointments for patients, given their illness. The system will categorize illness into a specialty, and set up an appointment with the right specialist doctor. Patients should not wait more than 14 days to see the doctor.

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. Given N > M, patients with certain illnesses will be assigned to certain doctors that may deal with different categories of illnesses.
3. The number of patients and information about each appointment will be recorded and tracked.
4. Depending on how serious the illness is, some patients may have to wait a few days to receive an appointment, but no patient under any circumstances must be made to wait for more than 2 weeks. The system will be designed so that there is a balance between how many patients are seen and how long a patient has to wait for an appointment.

Part B) Functional & Non-Functional Requirements:

Functional Requirements:

- A user will be able to enter illness information into the system and receive an appointment time and doctor.
- The system shall schedule appointments for all patients with doctors of the correct specialty.
- The system will ensure no patient has to wait more than 14 days to receive an appointment.
- The system will go through an additional check and provide a backup system to ensure all patients are seen in accordance with the severity of their illnesses.

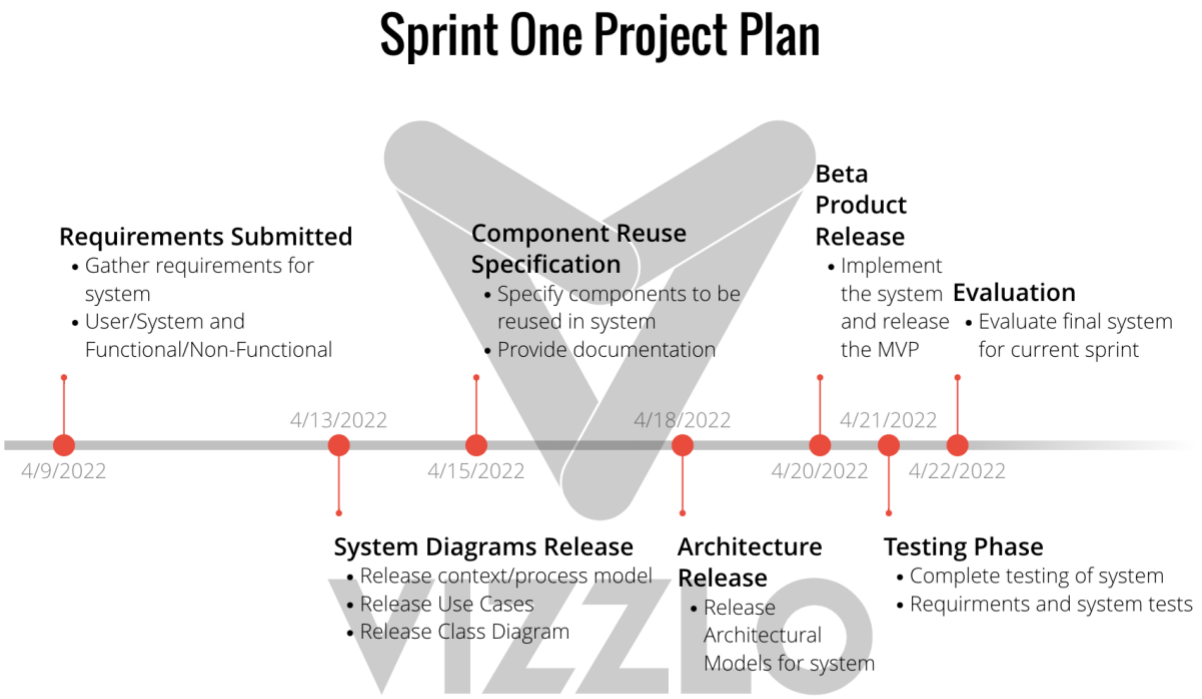
Non-Functional Requirements:

1. Product Requirements:

    a. Performance: Within a certain time frame, the system will return all patients to designated doctors. Users may expect several patients to be dealt with in a timely manner, so the system must be designed for optimum performance.
    b. Space: The system should take up the least amount of space possible and be as efficient as possible in matching every user (patient) to its specified doctor.
    c. Dependability/reliability: Given variable inputs (multiple patients, doctors, waitlist), the system must be built to accommodate everyone.
2. Organizational Requirements:
    a. Operational: The system will make sure under no conditions that a patient be left unattended for more than 14 days.
3. External Requirements:
    a. Ethical considerations: Without the user's permission, the system may not access personal data.

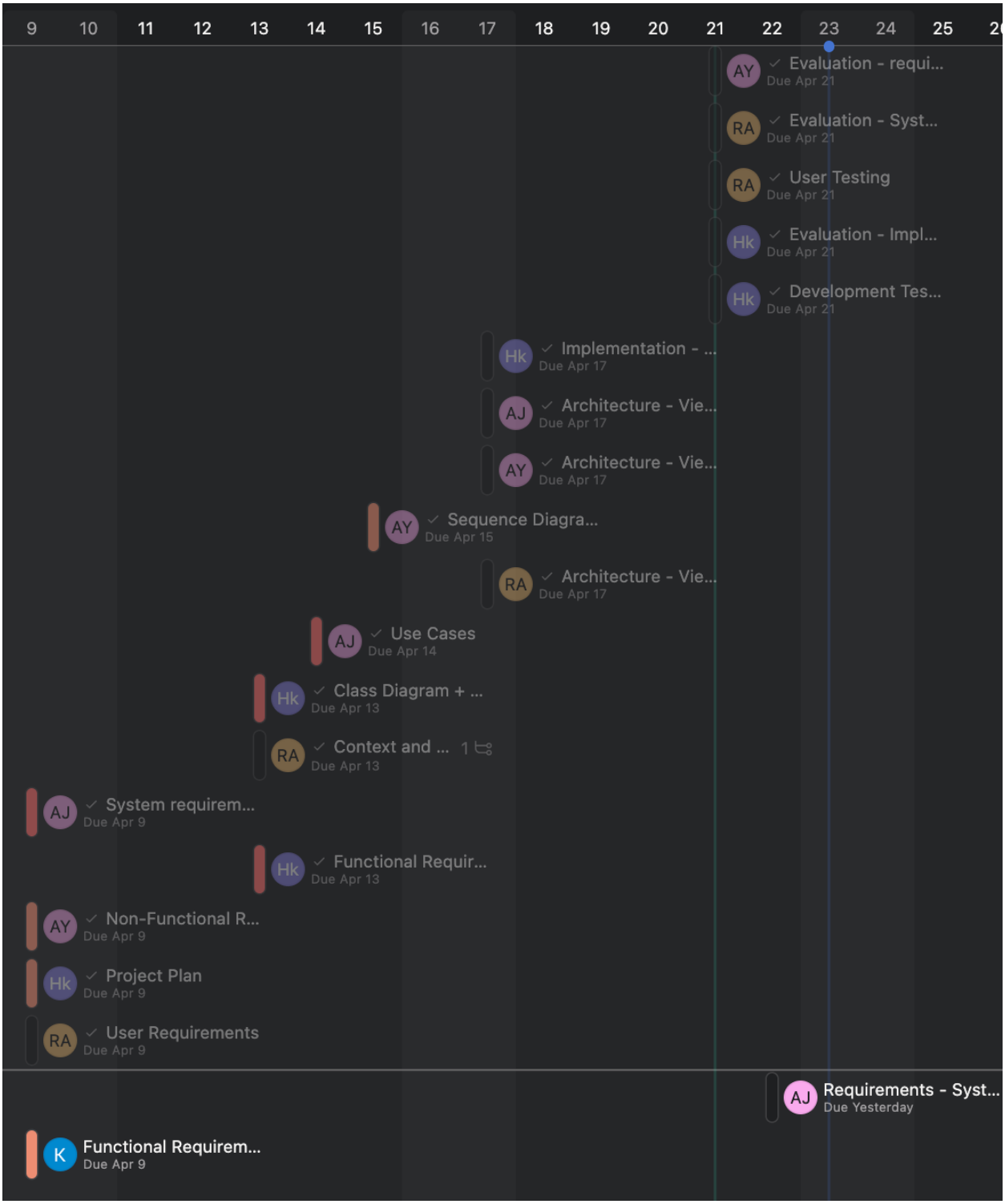
Project Outline Plan:

The project will consist of two sprints, where the first sprint will contain seven activities, and the second will contain eight (not including the project plans). For each sprint, a set of requirements, system diagrams, architectures, code, testing, dependability, reuse of components, and evaluation will be provided. Each activity will be considered deliverable to the customer, and each sprint will culminate with a final minimum viable product, along with some documentation (including requirements, testing, evaluation, etc…). The expected schedule for all deliverables is given below:



The project was managed effectively using the Project management tool, Asana. Asana allows the project to be broken down into activities, and then for each activity to be broken down into a set of tasks, to be delegated to each team member. The tool also tracks the progress of tasks by team members, allowing members to update their progress as they are completing tasks. A depiction of this software in use is given below:

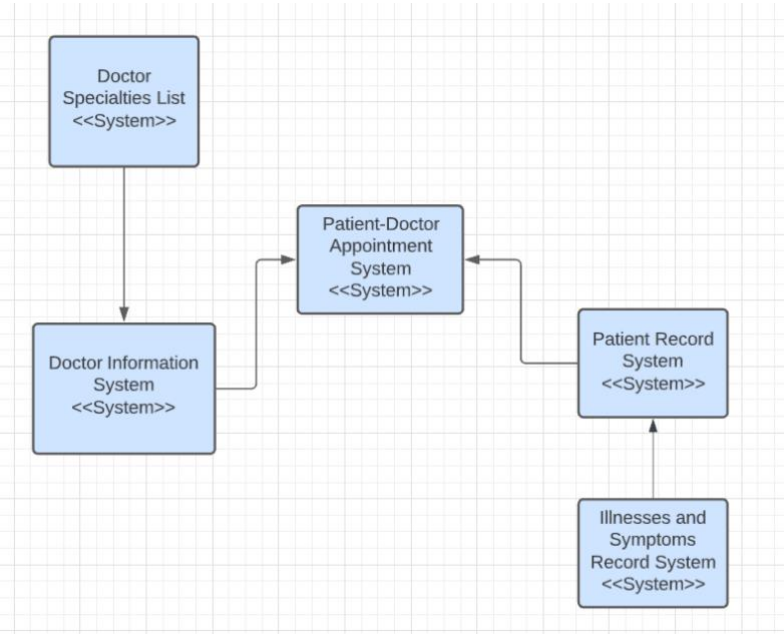
The project was split up into multiple tasks, across different activities (requirements, system diagrams,ect…). All tasks were assigned and tracked and progress was also updated using the Project Management tool Asana. The tasks, along with their dates and team members assigned to are given below (AJ, AY, RA, HK, K) represent the initial of the team members.


9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
AY Evaluation - requi... Due Apr 21
RA Evaluation - Syst... Due Apr 21
RA User Testing Due Apr 21
Hk Evaluation - Impl... Due Apr 21
Hk Development Tes... Due Apr 21
Hk Implementation - ... Due Apr 17
AJ Architecture - Vie... Due Apr 17
AY Architecture - Vie... Due Apr 17
AY Sequence Diagra... Due Apr 15
RA Architecture - Vie... Due Apr 17
AJ Use Cases Due Apr 14
Hk Class Diagram + ... Due Apr 13
RA Context and ... 1 Due Apr 13
AJ System requirem... Due Apr 9
Hk Functional Requir... Due Apr 13
AY Non-Functional R... Due Apr 9
Hk Project Plan Due Apr 9
RA User Requirements Due Apr 9
AJ Requirements - Syst... Due Yesterday
K Functional Requirem... Due Apr 9

System Diagrams:

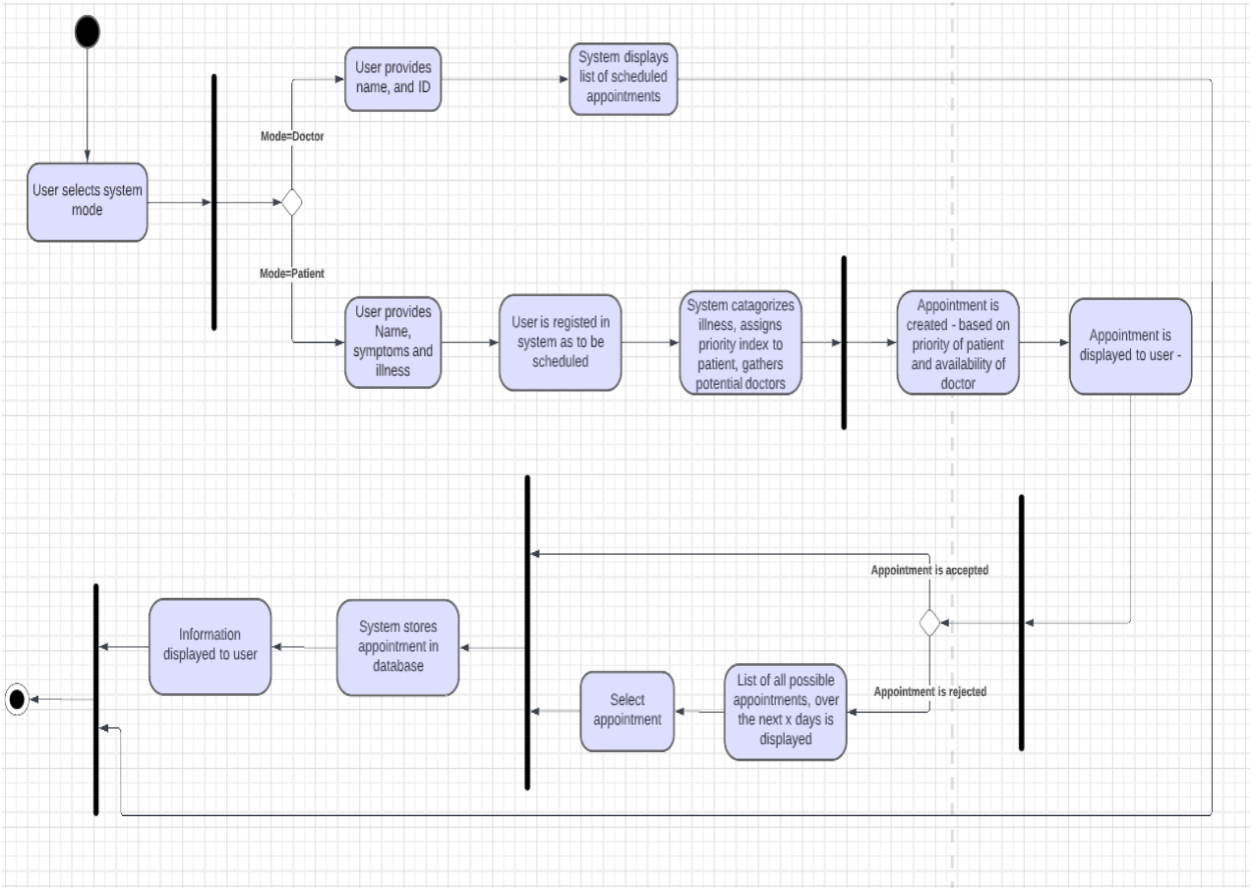
Part A) Context and Process Models

Context Model:



The system shall center around the sub-system which generates appointments for the patient with the doctor. When the appointment is created, the patient's personal information will be entered into the records. The patient's specific illness and the particular symptoms they are suffering will be noted down as well. Personal information and the specialty of the doctor make up a subsystem as well.

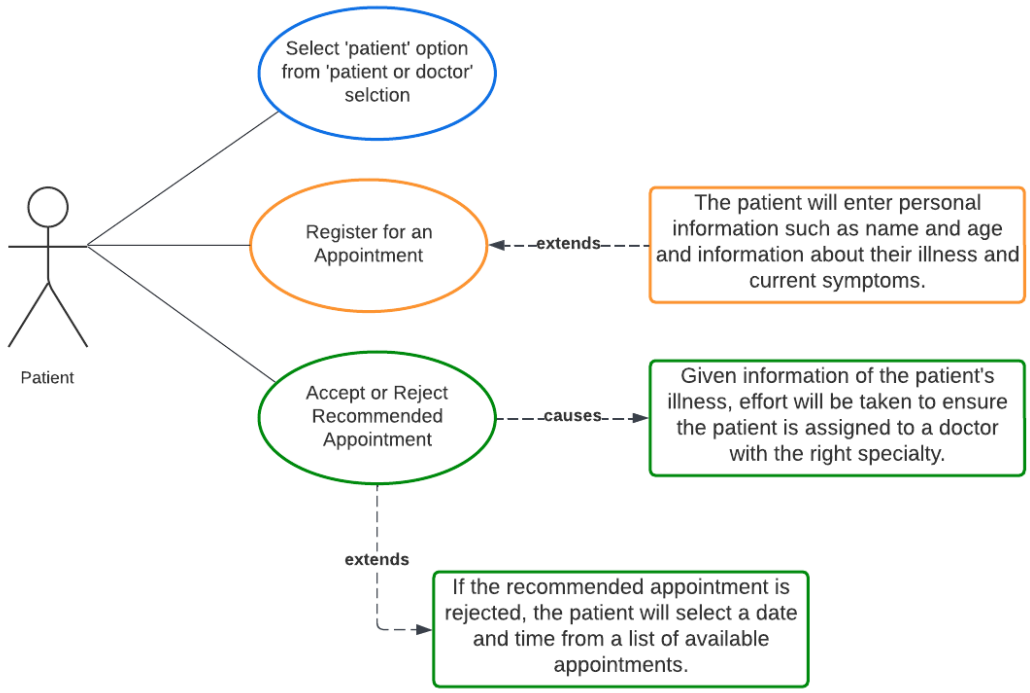
Process Model:



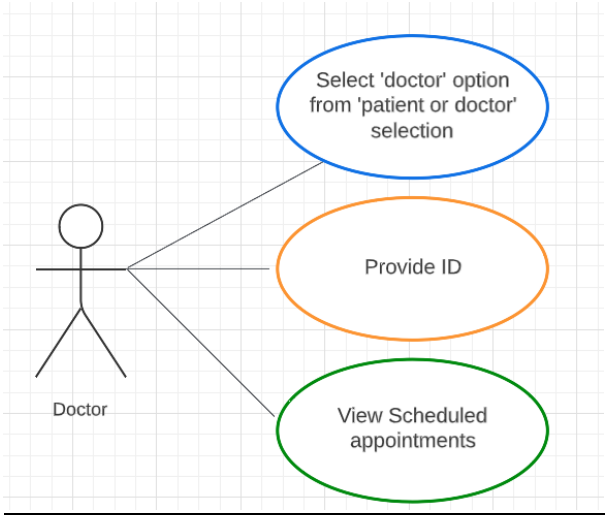
The filled circle indicates the start of the process. At first, the user selects the system mode (doctor or patient). If the selected mode is ‘doctor’, then the user will be asked to enter their name and ID. After that, the system will display a list of scheduled appointments with the doctor. However, 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 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 Case and Interaction Models:

Use Case Diagrams:

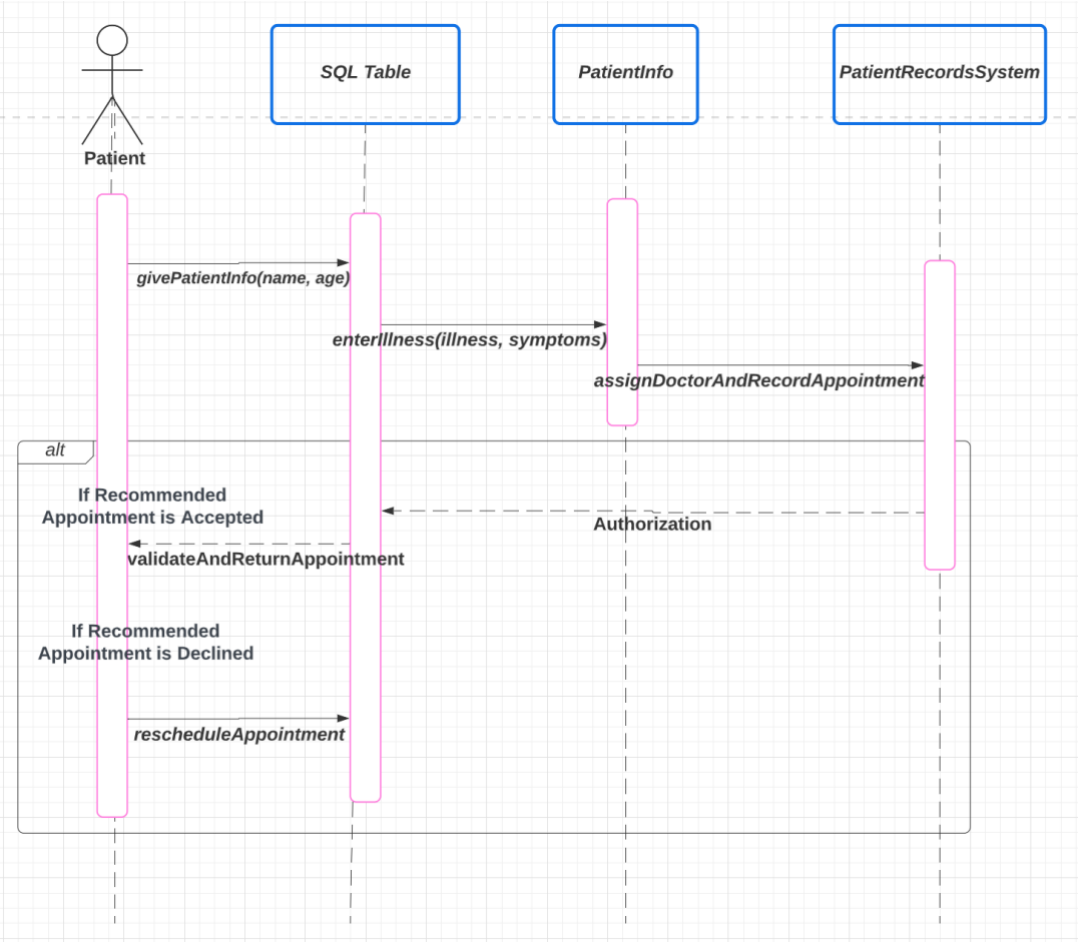


This use case diagram shows the perspective of a patient and how they will interact with the system. First, the user will select the ‘patient’ option from the two options: “patient or doctor”. The user will then input their name and register for their appointment, entering data about their illness. This ensures that the information given will allow them to be assigned with a doctor that can take care of their illness and be appointed with them as soon as possible. Lastly, the patient will be able to select an appointment from a list that the system will display to them if they first decline the appointment recommended by the system.

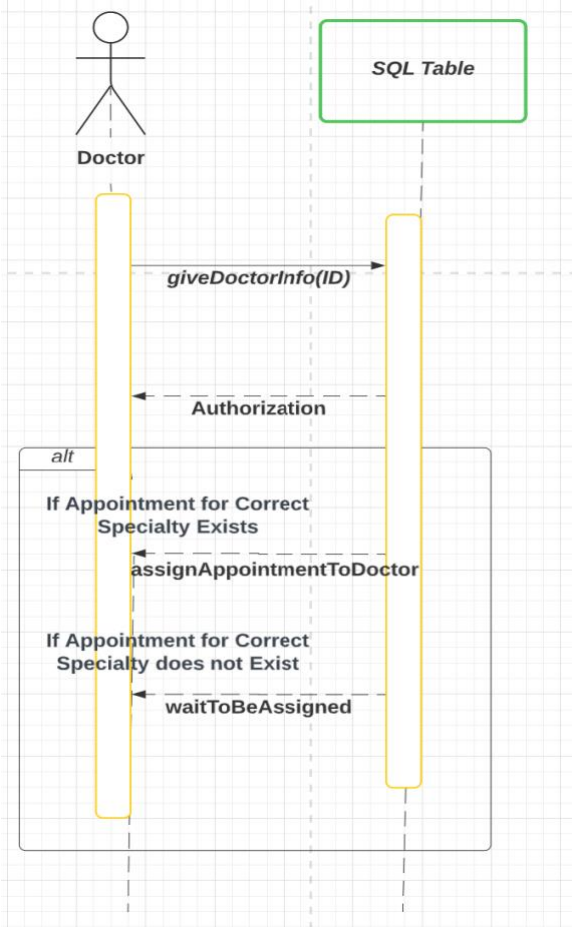


This use case diagram shows the perspective of a doctor and how they will interact with the system. First, the user will select the ‘doctor’ option from the two options: “patient or doctor”. Then, the user will input their ID, allowing them access into the system. Lastly, the user can view the appointments from the given list that the system will show.

Sequence Diagrams:

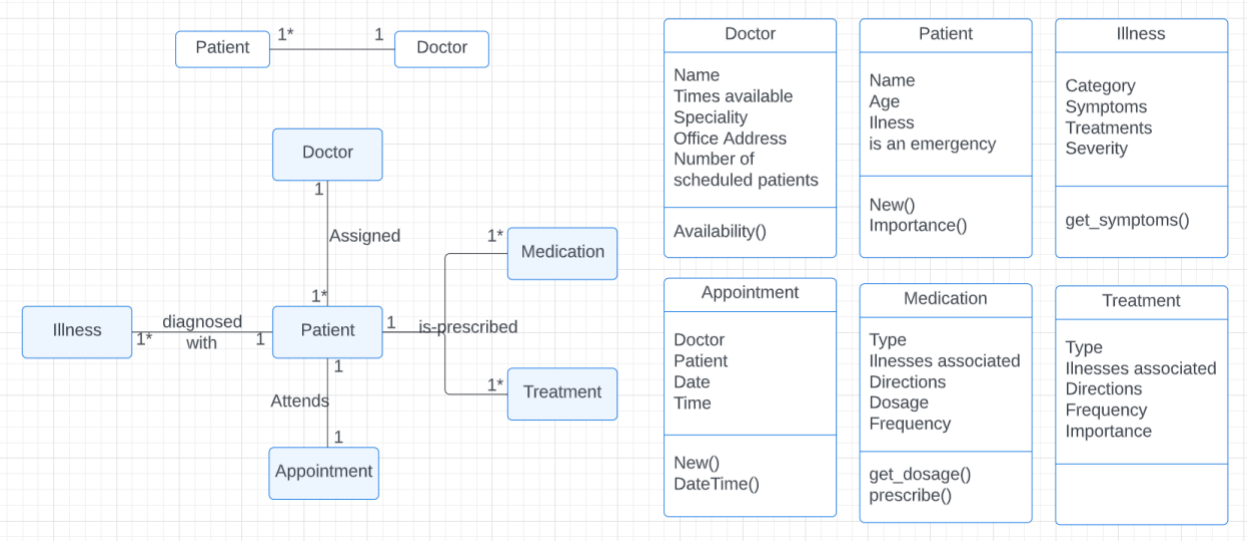


This sequence diagram portrays the use case of a patient using the system. They input their name and age to the SQL Table. Then, the system registers the patient's illness into the records and assigns an appointment with the specific doctor. If the appointment is accepted by the patient, the appointment is validated and recorded. If declined, the patient will reschedule an appointment by selecting from a list of appointments..



This sequence diagram illustrates the use case of a doctor using the system. First, the doctor inputs their ID. The system then authorizes the list of patients with the specific illness the doctor specializes in and matches appointments based on priority. If the patient list does not match the specialty, the patient gets waitlisted to be assigned an appointment with the correct doctor.

Part C) Class Diagram:

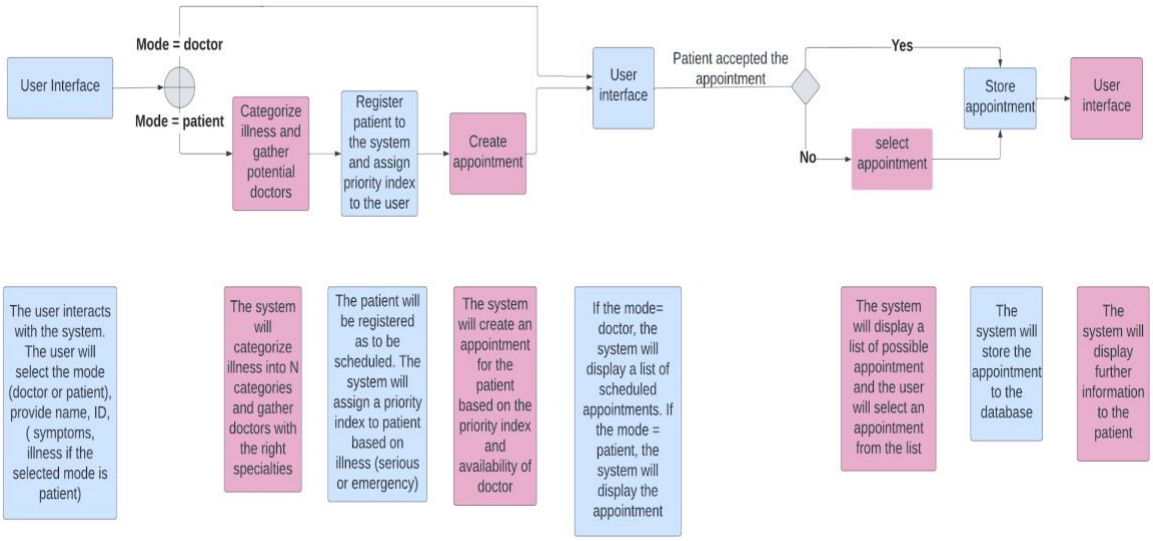


The class diagram depicts the real relationships between the different pieces of the system. The system focuses on creating a patient-doctor appointment system, with two main users being the patient and the doctor. The patient enters the system, sets the mode, and various inputs, and the system culminates by registering the patient, creating an appointment for them, and printing the details to the user. If the user selects the doctor as the mode of use, the user enters their name, and the system prints out their schedule based upon the doctor ID stored in the database (cross-referenced with the name inputted). The class diagram depicts the doctor and patient class, where one patient is assigned one doctor, but one doctor can have multiple patients. The remaining four classes focus on relationships with the patient class. A patient is diagnosed with an illness, or more than one illness, and is prescribed either one or multiple medications and treatments. Finally, the patient attends an appointment, where one patient can only attend one appointment.

The attributes and methods of each class are given in the UML diagram, depicted to the right. Briefly explaining the attributes of each, the doctor class contains a name, unique ID, and times available attribute, while the patient class contains a name, age, and illness attribute. Illnesses also contain a category and symptoms attribute, where the category is the classification of said illness, and symptoms are a list of common symptoms associated with the defined illness. The appointment also contains doctor and patient attributes (specifying the attendees of the appointment). Illness and medication classes are also given, with relevant attributes.

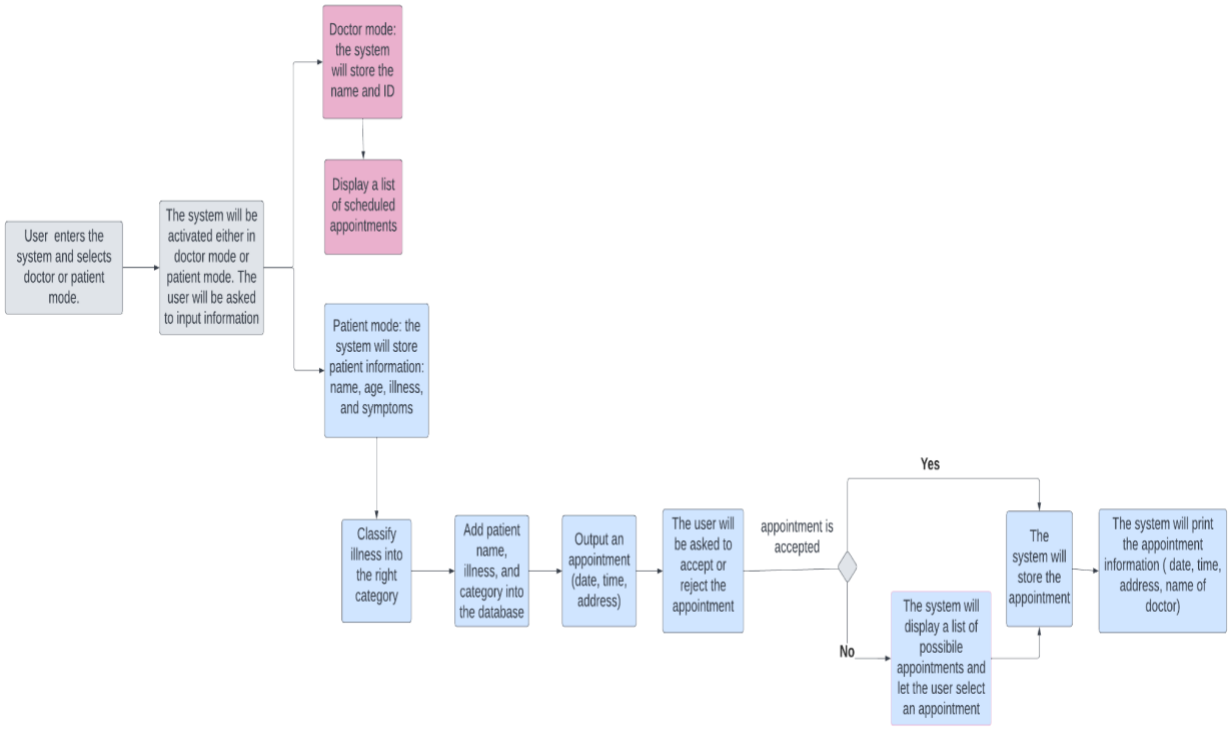
Architecture:

Part A) Architecture - Process View:



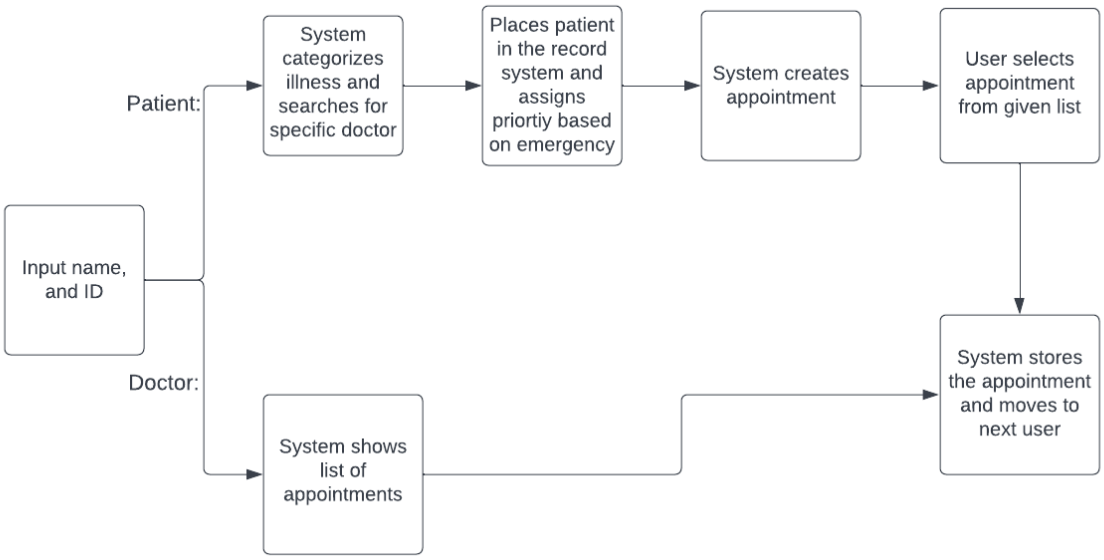
This architecture focuses on the overall system from a process view. The architecture will consist of a user interface to gather information from the user (doctor or patient mode, name, ID, etc.). Based on the entered mode, the system will go into one of two branches, doctor or patient. If the selected mode is patient, the system will categorize illness into N categories and gather doctors with the right specialties. Then the system will register the patient as to be scheduled and assign to him/her a priority index based on illness (serious or emergency). After that, the system will create an appointment based on the priority index and availability of doctors. Then the system will display the appointment to the patient. If the selected mode is doctor, the system will display a list of scheduled appointments. The patient has the option to accept or reject the appointment. If the user accepts the appointment, the system will directly store the appointment in the database. However, if the user rejects the appointment, then the system will display a list of possible appointments and the user will select an appointment from the list. Then the system will store the appointment in the database. Finally, the system will display further information to the patient.

Part B) User Interface:



This architecture focuses on the user interactions with the system. First, the user enters the system and selects the mode, doctor or patient. Then, the system will be activated based on the selected mode, and the system will ask the user to provide information. If the selected mode is doctor, the system will ask the user to input their name and ID, then the system will print a schedule of the doctor's appointments with different patients. However, if the selected mode is patient, the system will ask the user to provide his/her name, age, illness, and symptoms. Then the system will classify the user's illness into the right category and add the patient's name, illness, and category into the database. Then the system will output an appointment (date, time, address, etc.). The patient has the option to accept or reject the appointment. If the user accepts the appointment, the system will directly store the appointment in the database. However, if the user rejects the appointment, then the system will display a list of possible appointments and the user will select an appointment from the list. Then the system will store the appointment in the database. In the end, the system will print the appointment information (date, time, address, name of doctor).

Behavioral View:



This behavioral view shows how the overall system will work from both the doctor as a user and the patient as a user. If the user is a doctor, he/she will enter their name and ID and the system will proceed to show a list of all the appointments. If the user is a patient, the system will go through a series of questions to figure out the illness the patient has and whether it is an emergency or not. With the answers being recorded into the patient's record history, the system will create a list of appointments and will allow the user (patient) to select based on their availability. Lastly, in both cases, whether it is the doctor or the patient, the system will store these appointments in the database.

Testing

Part A) Development Testing - Automated tests

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

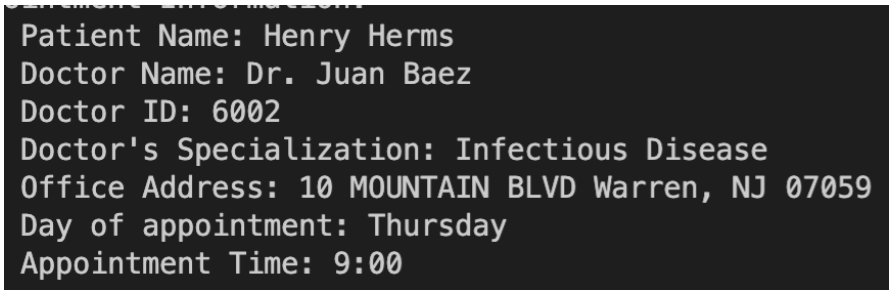
   For these sets of tests, we use the partition testing method, partitioning the system into multiple groups, where each group has a set of common inputs. We then create multiple tests for each of these groups. After executing the tests, we can begin to validate the result and determine whether the system 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.

   Test One:

   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.



   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 outputted 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 dont remember”, illness = “Schitzophrenia”, is_emergency=False,“No”,”Im tired”

The system returns an error, where there is an issue between the types of variables used in an if statement (str and int instead of int and int). This error and halt to execution are what we initially expected, as the age input is invalid, but the system should have instead checked in input prompted the user for valid input, in this regard it did not pass completely.

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,”Im tired”
For this test, the system provides an error as the patient component attempts to use the inputted illness and find the illness category in the internal dictionary. Since it does not exist, the program stops execution and returns an error, as expected validating the result of this test. In the name of dependability and better

software engineering practices, however, we should, in the future perform input validation and check to ensure the illness entered exists in the dictionary - before accepting the input from the user

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 seemingly invalid inputs would allow us to test the appointments components and validate the result

Inputs: Patient object(name,25,"Diabetes",False,""), doctor ID="6001", doctor speciality = "Family Doctor"

For this test, we alter the doctor id to a string instead of an int and provide a bounds doctor specialty. In this case, the system functions as normal, which was unexpected - as we had initially believed the system would return a type error where it requires an int() data type. However, the system, and in particular the appointment objects and components, were able to handle edge cases where the specialty is not listed, and it was also able to handle a unique doctor_id input. We do note one issue with the output, as the outputted doctor is a cardiologist but is printed out as a Family doctor. Although this is invalid, it is because we altered the value of an internal attribute in one of our tests, as a result, the output was changed - which is expected. This test can be considered passed, as the system handled a different input type.

Test Four:

Since one of the biggest components in the system is the database, we can design tests to alter the parameters which are used in the databases, to an invalid input format. The biggest parameters used are the doctor id, and the date and time variables supplied. As a result, we attempt to get the schedule of a doctor with an unrecognized id. This test fails just as expected, and even provides us with incorrect information - which must be changed for future use

Inputs/Changes: doctor id = 4543 (invalid entry)

For this test, we focus on supplying an invalid int doctor id and determining how the system responds. We expect an error to be returned and execution to end, but in the ideal case, the system would halt execution to determine that the input is invalid and pause execution until the user supplies a valid input. The system behaves as expected and the test causes the system to return an error where it is not able to find the doctor with the id provided, in the database. For future

systems, we should poll the user for a valid input and halt execution until one is provided - instead of returning an error and ending execution. This test also fails just as we had anticipated.

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 = 12:30pm

For this test, we expect the system to provide incorrect date/time information, as we are updating an attribute of an instance of the appointments class (as opposed to supplying inputs to functions). But we also expect the system to attempt to create an appointment with the supplied data, and return an error to the user where it cannot find the requested column. When we run the rest, we find that the system behaves as expected and returns an error to the under where it cannot find the queried column inside of the table. This test fails, as expected, and we must implement systems to validate inputs - to prevent this test from failing

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.

Test One:

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 provide an invalid date, which causes the system to throw an error and end execution, as we had anticipated. As a result, the system outputs some information about an appointment but is not able to complete requests to the database to add and create new appointments based on the date/time supplied. This test failed, just as we had expected.

Test Two:

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, where we expected that the system would halt execution and request data from the database with an invalid query. Since the date and time attributes go hand in hand, we must perform some input checking on both attributes before accessing the database with them. As a result, we can consider this test to have failed, in-line with our initial expectations.

Test Three:

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. While this is an edge case (since the date/time inputted are not given in the list of available appointments), it's important we adapt the system during the next iteration to ensure this case is covered (and this test is passed). We can consider this test to have failed, noting that we had not expected it to fail.

Test Four:

We can create one final system test, by providing valid inputs that are defined as beyond the specified bounds of the inputs. This means we will request a

date/time from the system that is not outputted under the list of available appointments

Inputs: date = Sunday, time = 5pm

This test also fails, as we had initially expected. The system attempts to update the gathered doctor's schedule in the database, by calling a column named using the date and time. Since there is no column present for Sunday, the system attempts to access a column that does not exist - causing an error to be thrown.

Part B) Release Testing:

System Requirements Tests:

1.) System Requirement One:

- A dictionary (illnesses_dictionary.py) has been created and filled with a list of numerous different illnesses and diseases. The specific illnesses are also classified into N more general categories (accepted_medical_specialities). Examples of these general categories include Respiratory, Neurology, and Infectious Disease.
- Testing enables one to see that if a patient inputs, for example, Asthma, this input will be stored in the string 'curr_illness', which is part of the patient object which gets stored in the database. Given this information, this illness is classified under a general category (Respiratory) using the illnesses.dictionary.py file. In this way the requirement of classification of illnesses is fulfilled.

2.) System Requirement Two:

- A file (doctors.db) includes a list of M medical specializations, and each doctor is classified under their particular specialties. M = 7, allowing that N > M. The medical specializations are: Cardiology, Respiratory, Gastro, Neurology, Psychiatry, Infectious Disease, and Endocrinology. The written code also takes care of the requirement that patients with certain illnesses are assigned to a doctor that deals with a particular specialty.
- According to Test One of Unit Tests, it is clearly seen for the test case of a patient inputting Tetanus as their illness, they are referred to a doctor who specializes in Infectious Disease (in screenshot above).

3.) System Requirement Three:

- The patient enters personal information (such as name and age) and information about their illness (such as if it is an emergency and any symptoms). This data is collected from the user, stored into instances of classes (Classes.py) while the

program is executing, and then recorded in multiple SQL tables, to ensure the system can access the data once the program has terminated.

4.) Depending on how serious the illness is, some patients may have to wait a few days to receive an appointment, but no patient under any circumstances must be made to wait for more than 2 weeks. The system will be designed so that there is a balance between how many patients are seen and how long a patient has to wait for an appointment.

- The code uses 6 classes for 6 days of appointments per week and based on the age and emergency of illness, the code categorizes which patients are top priority and assigns appointments accordingly. No patient will be unattended for more than 6-7 days according to the system.

Functional & Non-Functional Requirements Tests:

1.) Is a patient able to enter illness information into the system and receive an appointment time and doctor?

- Test Five:
  Inputs: mode = “Patient”,name = “”,age =40, illness = “Coronary artery disease”,is_emergency = False, symptoms = “”, apointment_accepted = True
- According to unit test five, it is evident that the system passes the requirement of a user being a patient and entering their illness and systems to receive an appointment with the specialized doctor.
- A patient is able to enter detailed data about their illness. Depending on this information, the patient is assigned to a doctor that deals with the specialty that the illness is categorized under. The patient is also informed of the date and time of the appointment.

2.) Does the system schedule appointments for all patients with doctors of the correct specialty?

- Test Three:
- Inputs: mode = “Patient”,name = “”,age =30, illness = “Diabetes”,is_emergency = False, symptoms = “”, apointment_accepted = True
- According to unit test three, it is evident that since the patient is facing the illness regarding diabetes, he/she is appointed with an endocrinologist which is the specific doctor who treats patients with diabetes. Thus, the system passes this requirement.
- Common illnesses are categorized under a general specialty, and each doctor has been classified according to their particular specialty. Therefore, the patient will be assigned to the correct doctor.

3.) Does the system ensure that no patient has to wait more than 14 days to receive an appointment?

- Once the patient enters their illness and symptoms, he/she receives a certain time and date for their appointment with the specialized doctor. If they would like a different appointment date/time, they receive a list of all the days and times available for the week and can choose whatever time works best for them.
- The system provides 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.

Part C) User Tests:

The system has passed all the unit tests. However, when the user provides that he/she has an emergency with a medium-low risk disease and a low-risk age, the behavior of the system is abnormal. The system did not provide the address of the doctor. The appointment date for this case is acceptable although the system should provide an earlier date since there is an emergency.

For component tests, the system returns an error when the user provides invalid inputs as expected. However, the system should check for invalid inputs and ask the user to provide a valid input.

The system provides an error when the user provides an illness not stored in the dictionary by the system. However, the system should check for the entered illness and ensure that the illness exists in the dictionary before accepting other inputs from the user.

The system returns an error when it is not able to find the doctor with the ID provided. However, the system should check for invalid ID and halt execution until the user provides a valid ID.

The system returns an error when the user inputs incorrectly formatted date and time. However, the system should check for the date and time format and ask the user to provide the date and time in the correct format.

For system tests, when the user declines the suggested appointment and tries to select an appointment, but the user provides an invalid date or time, then the system returns an error and ends execution. The system should be altered to check for the provided date and time before accessing the database with them and ask the user to provide valid inputs, so the system is able to complete the request and create a new appointment.

When the user provides valid inputs, but they are beyond the specified bounds of the inputs the system will continue executions and return an error at the end. The system should check for the

entered inputs and ensure that they are within the specified bounds of inputs before accessing the database with them and ask the user to provide inputs within the specified bounds of inputs.

Overall, the system is still functional and is able to create appointments and provide a patient with appointment information. A doctor is also able to get a schedule of their appointments with different patients. I would recommend releasing the system as an initial version. All the issues and edge cases mentioned above should be solved in the second sprint. In addition, for the second sprint, it is important to ensure that appointments that are scheduled with doctors are not overwritten when the user inputs date and time occupied by another patient. The system must ensure that two patients must not get the same appointment at the same time.

Evaluation:

Requirements Engineering

- Are all requirements listed clearly?

  All user, system, functional, and non-functional requirements are elaborated precisely and support the design and intended purpose of the system. The requirements that have been listed out provide sufficient detail for the system to function properly.

- Easy to understand and written thoroughly?

  The requirements are easily understood and written clearly. The clearness allows for precise and correct coding implementation.

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

  Yes, the basic functionality of the system is consistent with requirements. When the patient enters specific information, the information is duly recorded; this data is then used to correctly pair a specialized doctor with the patient. Databases of illnesses and general medical categories have been created to take care of the requirements that the amount of possible illnesses is greater than the amount of medical specialties. Also, each patient is assigned a priority index, allowing patients with emergencies to be seen first. Assigning priority helps to arrange appointments in a fast, efficient, and orderly manner, thus fulfilling the requirement that no patient should have to wait more than two weeks for an appointment.

System Modeling

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

The system is represented by different types of models (Context and process models, use case diagrams, sequence diagrams, and class diagrams). Each model represents the system from a

different aspect. The context model is a high-level block diagram that shows different systems used with the Patient-Doctor Appointment System. The process model provides an abstraction of the system process. The use case diagram shows the system's actors (patient and doctor) and their interactions with the system. The sequence diagram shows the interactions between the user (patient or doctor) and the objects in the system. The class diagram represents the real relationships between the different pieces of the system.

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

Each diagram represents a clear understanding of the system from a different aspect and shows clearly how the system will operate. The context model shows that the system will use a doctor information system, patient record system, and illness and symptoms record system. The process model shows that there will be two operation modes (doctor and patient) and the system will be activated based on the selected mode. The system will ask the user to input some data and the system will provide a doctor with a schedule of their appointments. The system will provide a patient with a suggested appointment and the patient can decline the appointment and select a new appointment from a list of available appointments. The use case diagrams show how the patient will interact with the system by letting the user select the patient mode and register for an appointment by providing information about their name, age, illness, and symptoms. The system ensures that the patient can accept or decline the suggested appointment and select a new appointment when the suggested appointment is rejected. The second use case diagram shows how the doctor interacts with the doctor by selecting doctor mode and providing their name, ID. The sequence diagram shows the interactions between the user (patient or doctor) and the objects in the system and in what order. The class diagram represents the real relationships between the different pieces of the system.

Design and Implementation

- Code is clear and easy to understand?

The code is for the most part clear and easy to understand. As the source code is considerably large, almost 1000 lines in total, it is difficult to ensure the source code is clear, easy to read/understand, and functionality sensible. Some of the code is redundant and elongated, for instance, multiple loops are used with if statements including over 6 branches, which can create some redundancy and make the code more difficult to follow. However, for the most part, functions are named well, according to what they do during execution, and all variable names are named correctly and are relevant to the values they store. While the code is somewhat long and redundant, it is extremely organized, where the different classes/attributes

and executable programs are classified according to their specification, and function (as detailed in the process model)

- Code works with no errors?

Since the code was developed in python, it is not possible to check if any errors occur during compilation, as the code is not compiled but rather it is interpreted by the python interpreter. As a result, errors are revealed during runtime and have to be handled individually, as they occur (unless using a debugger), As errors vary based on the user commands/inputs, it is difficult to determine whether the code will work in 100% of its cases with no errors. However we can confidently say that the code works (assuming rules are followed) with no errors, unless the user attempts to do something or provide something beyond the scope/limits of the system.

- Variables and functions are used correctly?

All variables and functions are used correctly, variables store temporary values based on their type (dictionary,boolean,int,string,...). Each variable is defined, and sometimes with its type to ensure that a type error is not raised during runtime. Variables are then fed into functions as parameters, and to external calls (specifically to grab/store/delete/update values inside of the database). Functions are also used correctly, as the system focuses on ensuring that each function has some defined action to do, based upon its inputs. And each function is used to partition the scope and process of the system into multiple blocks that can be easily intercepted by other functions/tested by the developers and engineers and controlled by other methods. The hierarchy and process of which functions are called, and in what order is also used correctly, in a methodological approach during the main function and testing function.

Architectural Design

- Are architectural designs available?

The architectural designs are available, and each architectural model depicts the system in a unique way. Conceptual view, behavioral view, user interface view, and process view are the architectural views used in this sprint. All are portrayed in detail regarding how the system will work. Below each diagram is an explanation of how they were derived and the thought process behind each of them.

- Designs are well organized and portray a clear understanding of how the system will work

Each architectural model depicts how the system should be organized and built from a distinct perspective, as well as a description of how the system will be organized and created. For example, the process view shows a broader understanding of the system with the interactions between the user interfaces. The user interface view shows how the user will interact with the system from both the doctor’s perspective and the patient’s perspective. The diagram goes through a detailed view of how the system will work portraying a better understanding of the

system as a whole. Lastly, the behavioral view shows another point of view of the system as a whole and how it will run and store information.

Software Testing

- Outputs as expected?

The necessary outputs of the system are properly produced by the software. When the driver is run, (and assuming the user is a patient), the patient can input all relevant data about their illness and expect to be recommended an appointment or given the opportunity to choose an appointment given availability of the doctors. The illness is categorized properly, and the patient is entered into a table which tracks patient information. Given the category and priority of the patient (which depends on symptoms and possible emergency), the specialized doctor is assigned and an appropriate date is set for the appointment. This ensures that patients who are suffering more seriously are seen first, and as each patient is assigned a priority index, the appointment outputs that are generated by the system are timely and allow for a maximum wait time of one week. Therefore, the outputs are produced as expected, hence helping the system to meet expectations and function smoothly.

- Multiple test cases are used?

Yes, multiple test cases are used to verify different scenarios. Both the patient and doctor interfaces are checked in order to ensure the data given by either patient or doctor is recorded correctly, allowing an appropriate pairing of patient and doctor for an appointment. Different information for the various variables involved are inputted during Unit Testing and Component Testing in order to confirm the system responds as it should for different illnesses/symptoms, different priorities, and whether the appointment is accepted or not. The system has been suitably built to handle these numerous possible test cases.