Medical Information Management System

UML Lovers:

Patrick Bush, Zach Chandler, Charles Jennings, Carson Jones, Christian Kauten

22 March 2016

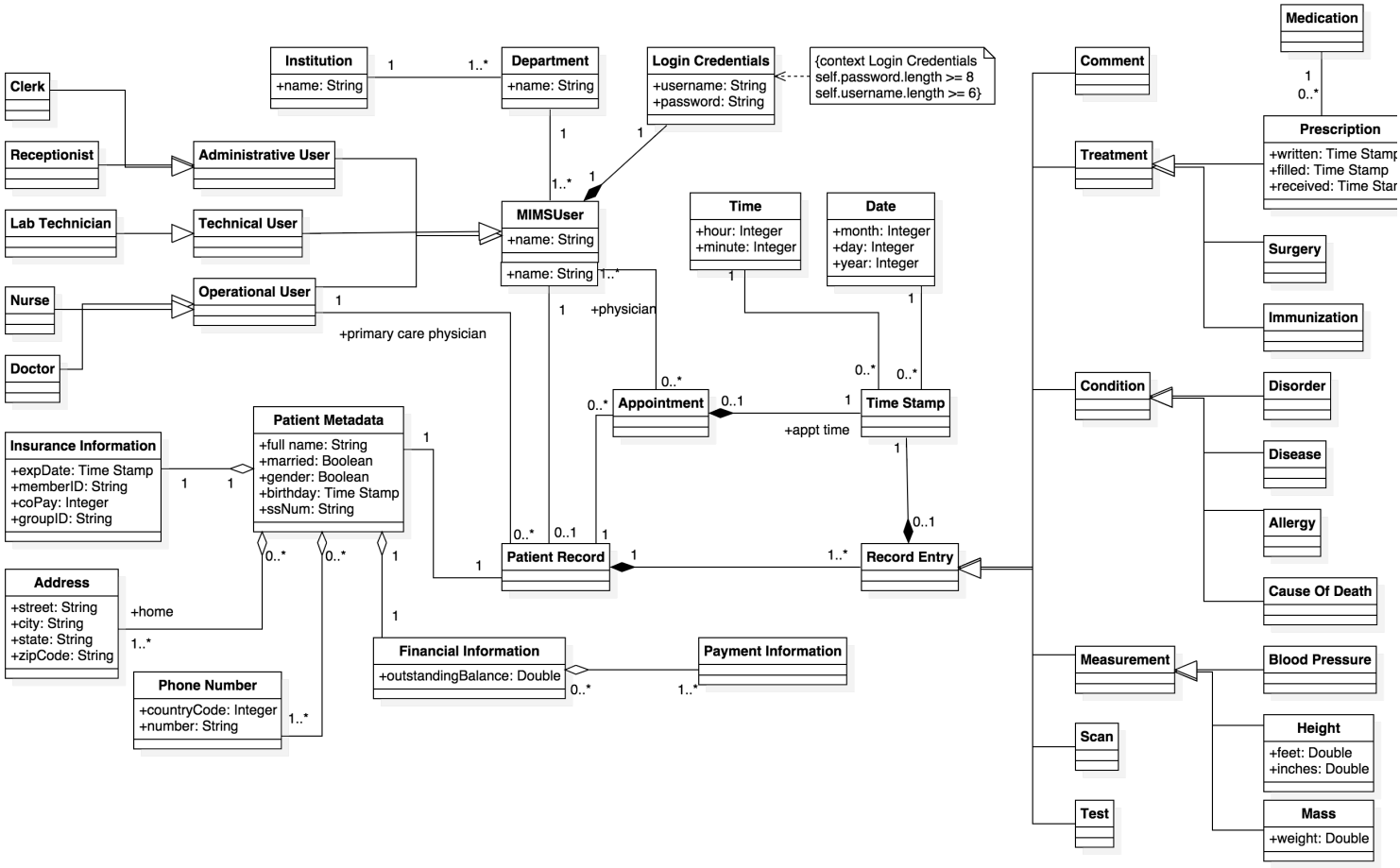
Contents

1.1 Concept Statement

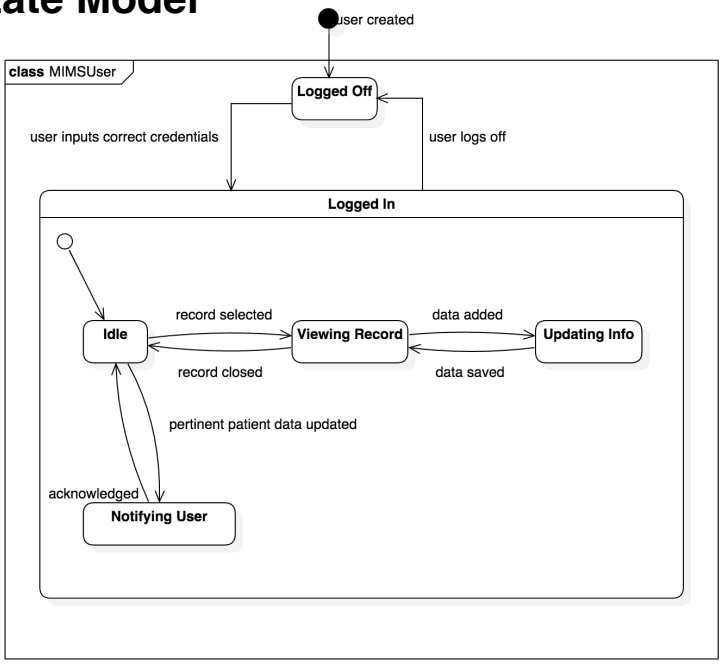
MIMS intends to streamline the transmission of patient healthcare records[1] between healthcare professionals. Each healthcare professional will be assigned a user profile from which they can interact with patient records. Profiles will be assigned security credentials in order to prevent unauthorized access. Records can be viewed by all users, but created, updated, and deleted only by select users. Medical personnel such as doctors and nurses have access to patient information such as treatments administered[2], measurements[3], diagnosed conditions[4], requested scans, requested tests and staff notes. Stay members like receptionists and clerks gather general information about new and returning patients upon entry. They will be able to create new files and edit basic patient information[4], upcoming appointments, which staff member is assigned to which patient, insurance information, and financial information[6]. They are also responsible for discharging patients. Finally staff members like lab technicians and the personnel responsible for collecting and tagging samples from patients (for tests and scans) will be able to access only information regarding these tests and samples. Staff members and departments[7] are notified when pertinent information in a file is updated. For example, a doctor is notified when results are back for a test they requested. All information will be recorded in real time and timestamped upon input. MIMS aims to help medical health care professionals deliver better care by: providing them with richly detailed medical histories of their patients, replacing tedious paperwork with a quick and simple software interface, and simplifying communication between departments[7]. Although not directly associated, MIMS intends to inadvertently aid patients by ensuring their healthcare professionals have access to their full medical history so they can receive the care they need.

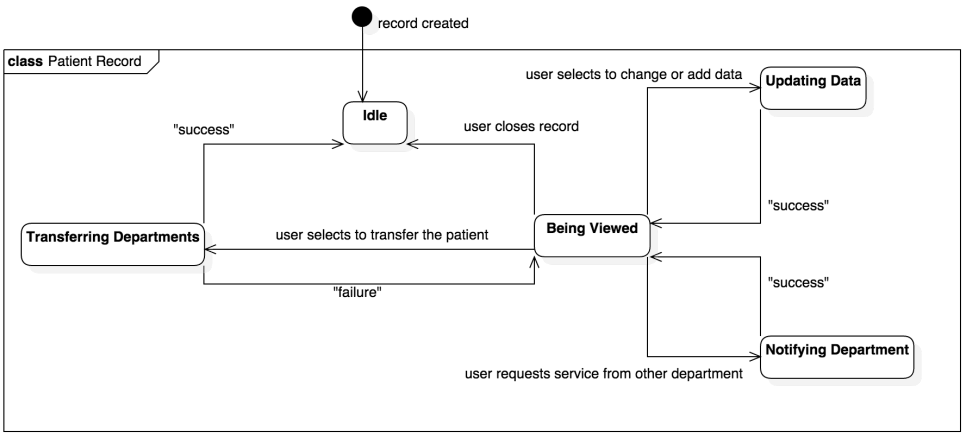
1: Name, Home address, Birthdate, Phone number, Marital status, Known allergies, Current medications, Primary care physician (PCP), Complaints and problems, Payment information, Requested tests, Test results, Requested scans, Scan results, Immunization records, Surgeries performed, Prescriptions written, Prescriptions filled, Prescriptions received, Current medications, Treatments administered, Allergies, Height, Weight, Birthdate, Blood type, Blood pressure, Diagnosed conditions, Upcoming appointments, Outstanding balances, Insurance, Notes
2: Treatments: Immunizations, Surgeries, Prescriptions written (Current medications)
3: Measurements: Blood pressure, Height , Weight , BMI, Pulse
4: Conditions diagnosed, Allergies, Diseases, Disorders, Death
5: Basic information: Name, Gender, Home address , Birthdate, Phone number, Marital status, Known allergies, Current medications, Name of primary care physician (PCP)
6: financial info: Payment info (credit card), outstanding balances.
7: Accident and Emergency (A&E), General Surgery, Laboratory, Neurology, Nutrition, Occupational Therapy, Pediatrics, Pharmacy

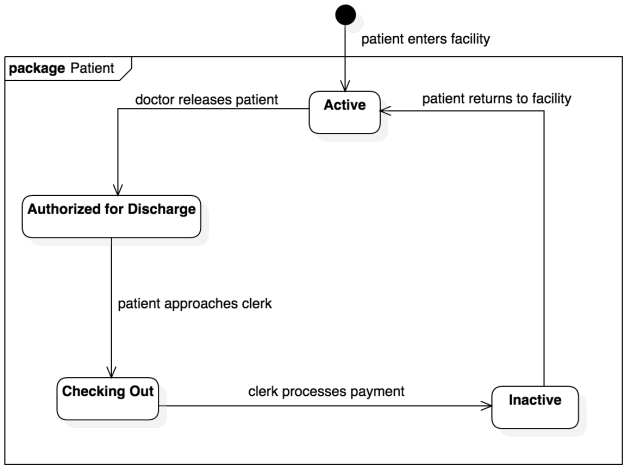
1.2 Conceptual Domain Model



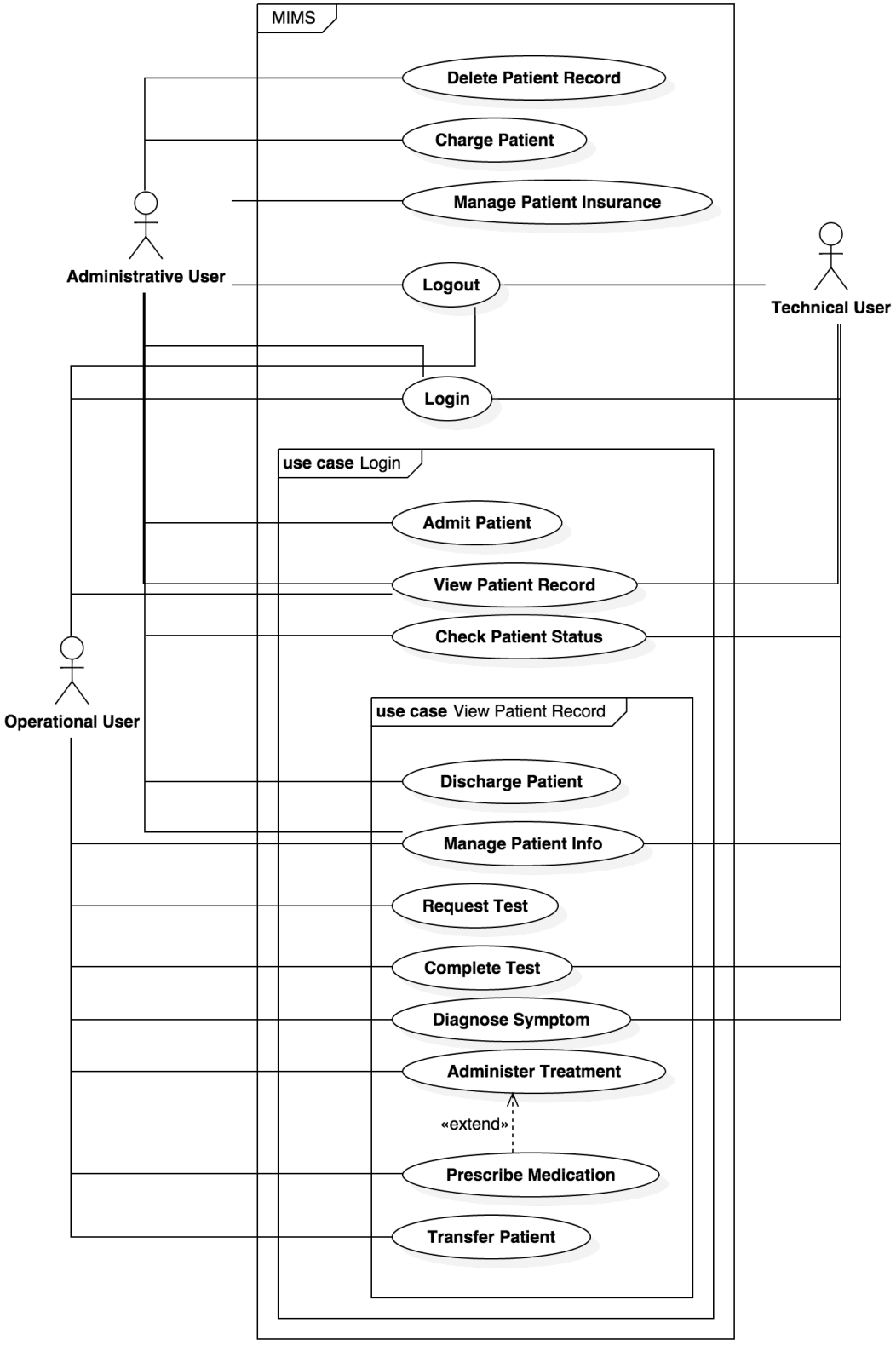
1.3 Domain State Model







2.1 Use Case Diagram

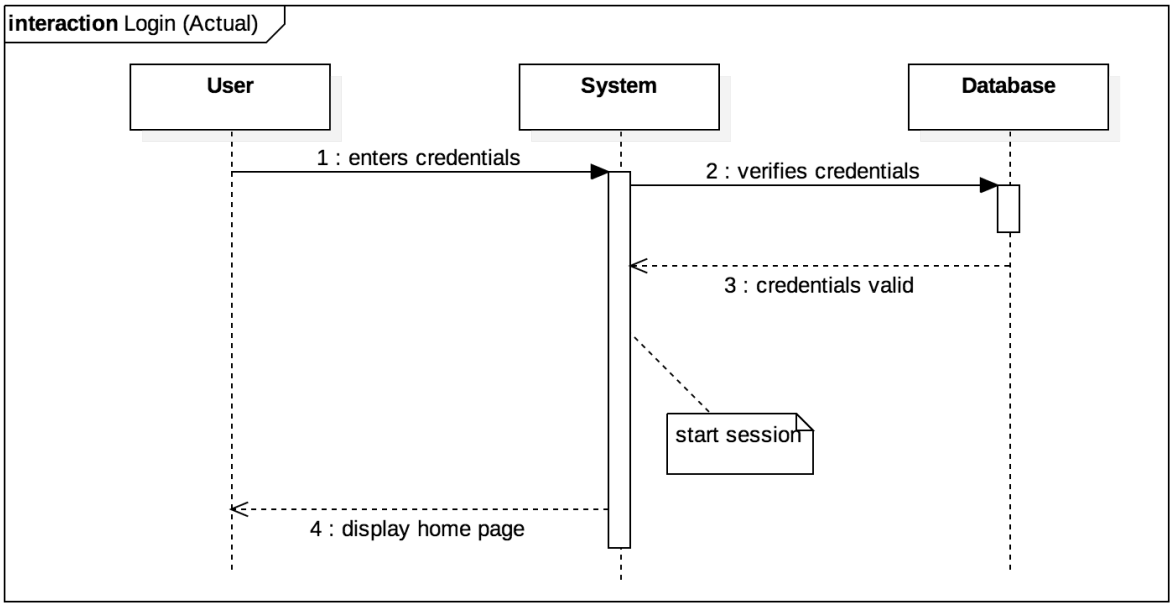


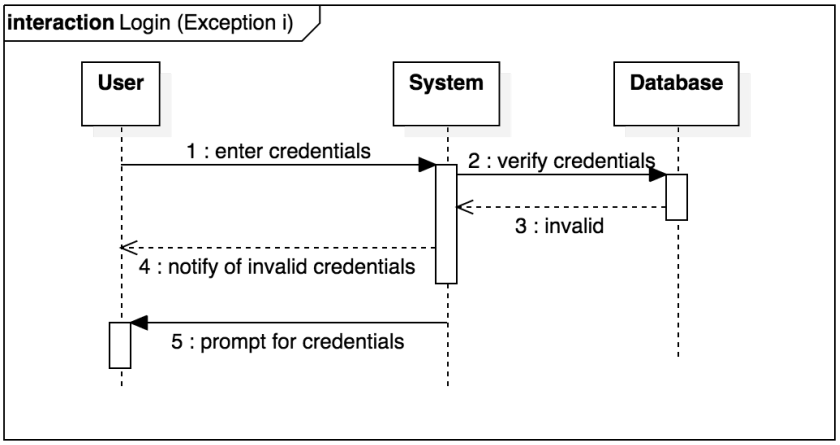
2.2 Application Interaction Model

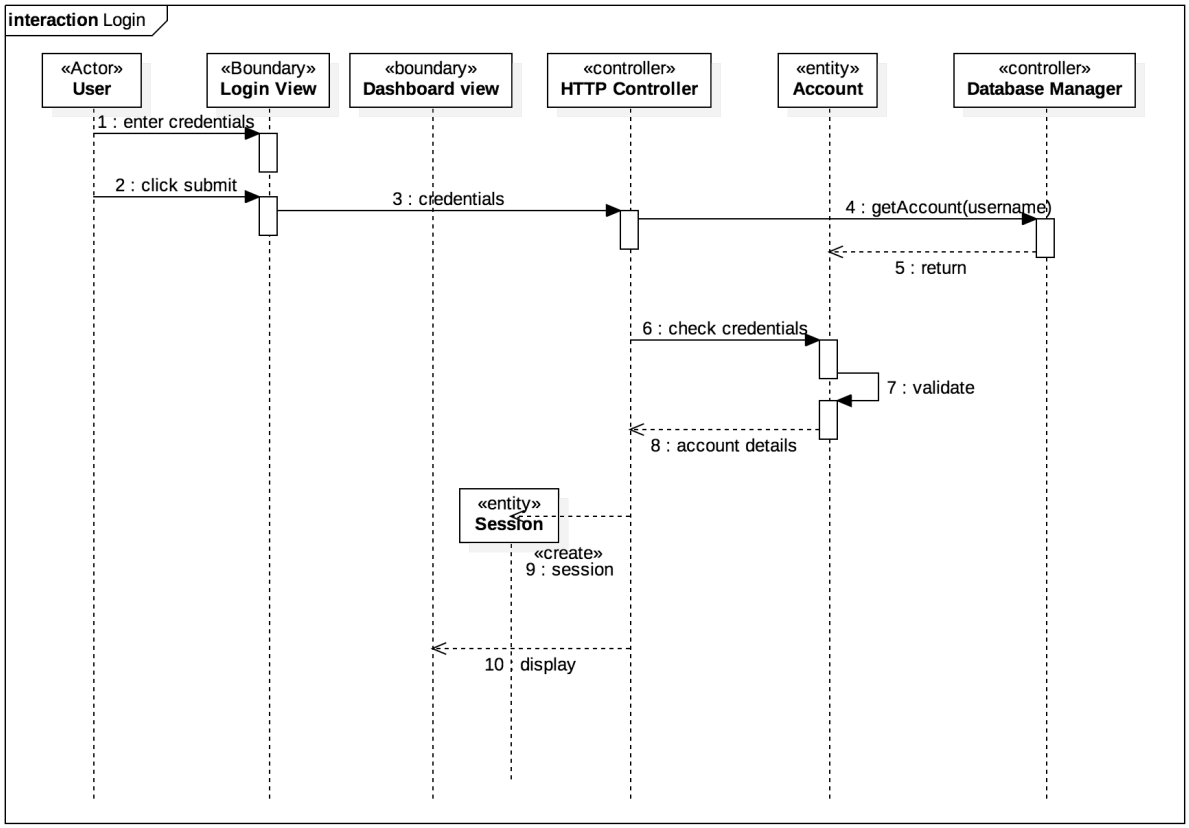
1. **Login**
    a. **Participants**: Administrative user, Technical user, Operational user
    b. **Precondition**: User has access to the system, with a valid username and password
    c. **Typical Course of Events**

Actor Intentions	System Responsibilities
1. User activates terminal	
	2. Terminal prompts user for their username and password
3. User enters their login information	
	4. Terminal verifies the username and password
	5. Terminal indicates successful login and allows the user access to the system

    d. **Alternative Courses**
        i. **Step 4**: User inputs an invalid username or password. Terminal indicates login error and prompts for username and password again.
    e. **Postcondition:** User is logged in


interaction Login (Actual)
User
System
Database
1 : enters credentials
2 : verifies credentials
3 : credentials valid
start session
4 : display home page

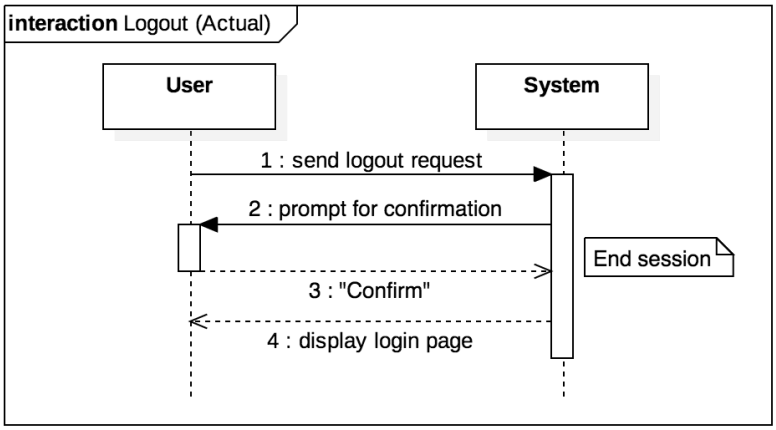

interaction Login (Exception i)
User
System
Database
1 : enter credentials
2 : verify credentials
3 : invalid
4 : notify of invalid credentials
5 : prompt for credentials

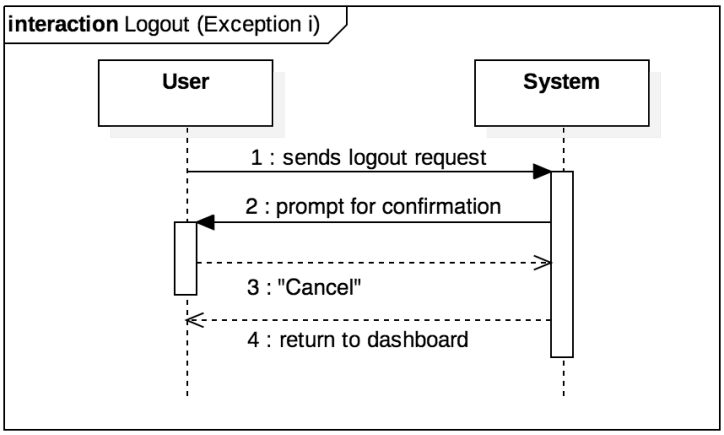

interaction Login
«Actor»
User
«Boundary»
Login View
«boundary»
Dashboard view
«controller»
HTTP Controller
«entity»
Account
«controller»
Database Manager
1 : enter credentials
2 : click submit
3 : credentials
4 : getAccount(username)
5 : return
6 : check credentials
7 : validate
8 : account details
«entity»
Session
«create»
9 : session
10 : display

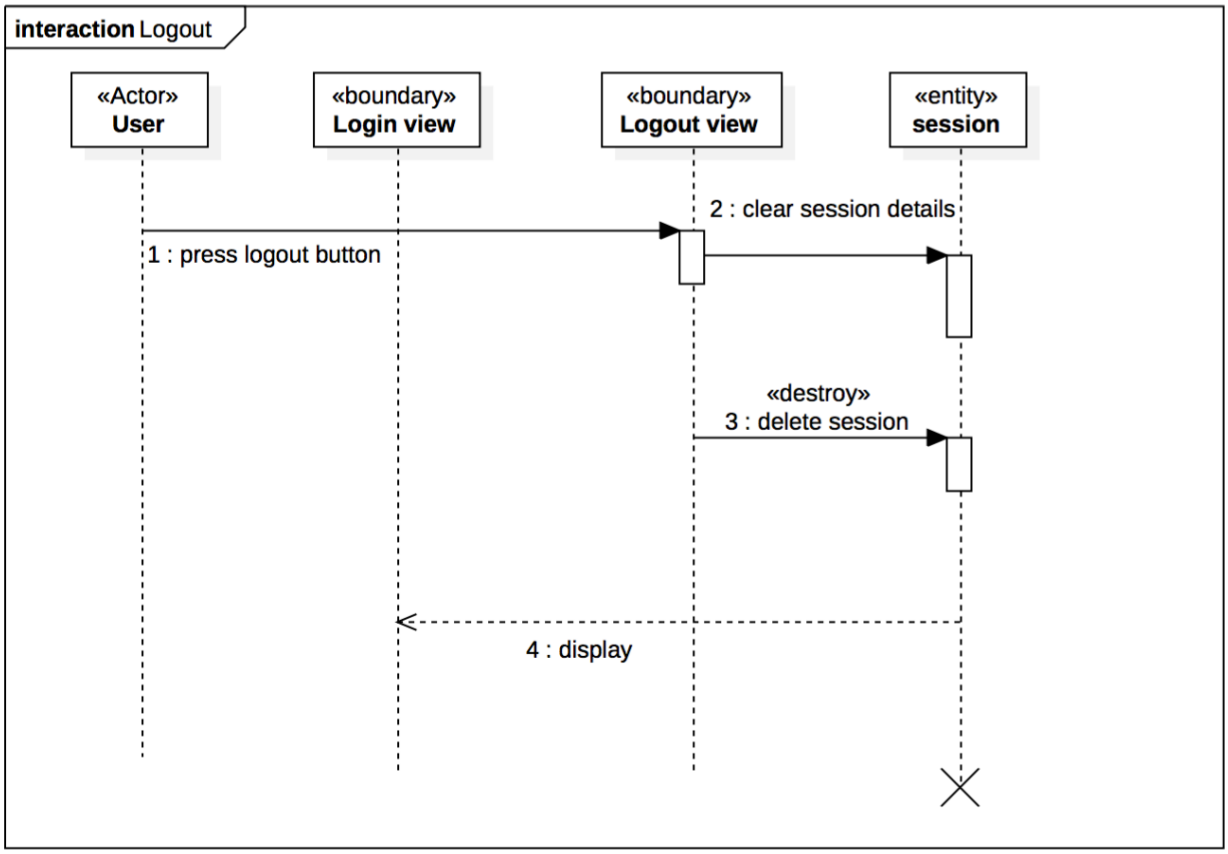
2. **Logout**
    a. **Participants:** Administrative user, Technical user, Operational user
    b. **Precondition:** The user is already logged in and is ready to log out of the system
    c. **Typical Course of Events**

Actor Intentions	System Responsibilities
1. User indicates they want to log out	
	2. Terminal prompts the user to make sure they want to log out
3. User acknowledges that they want to log out	
	4. System invalidates the user's session
	5. System displays successful log out message and clears any user information that was part of their user session

    d. **Alternative Courses**
        i. **Step 3:** The user indicates that they don't actually want to log out
    e. **Postcondition:** The user is now logged out of the system and the system is ready to be logged into again



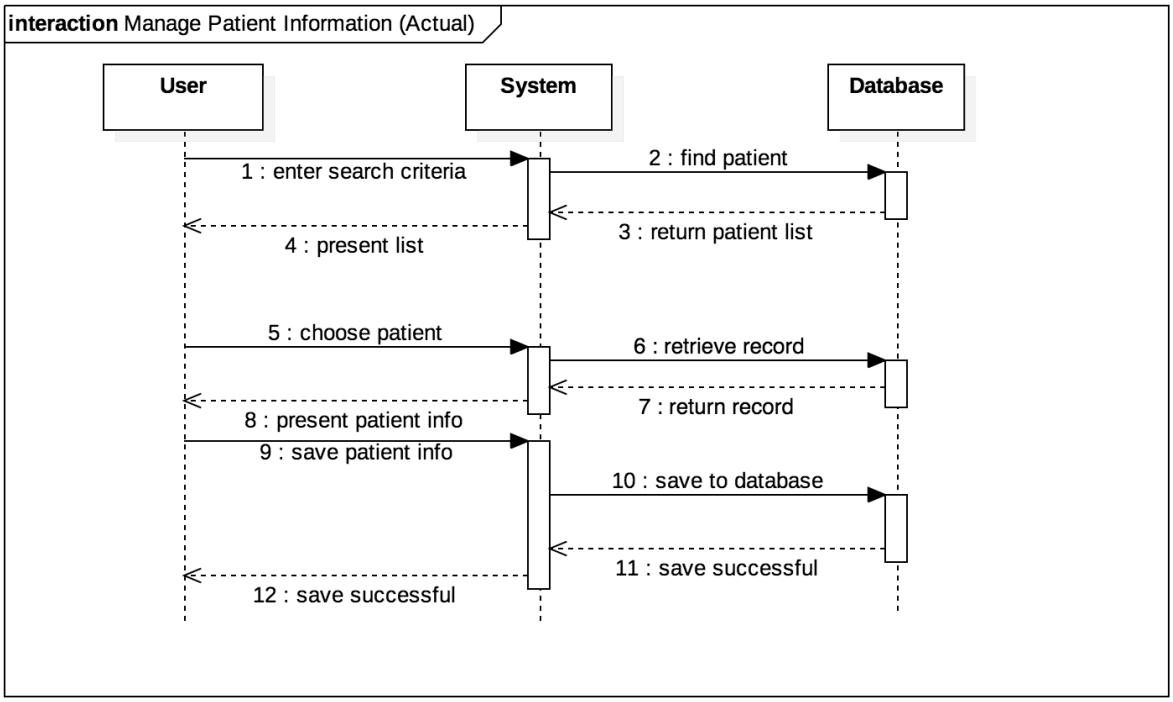


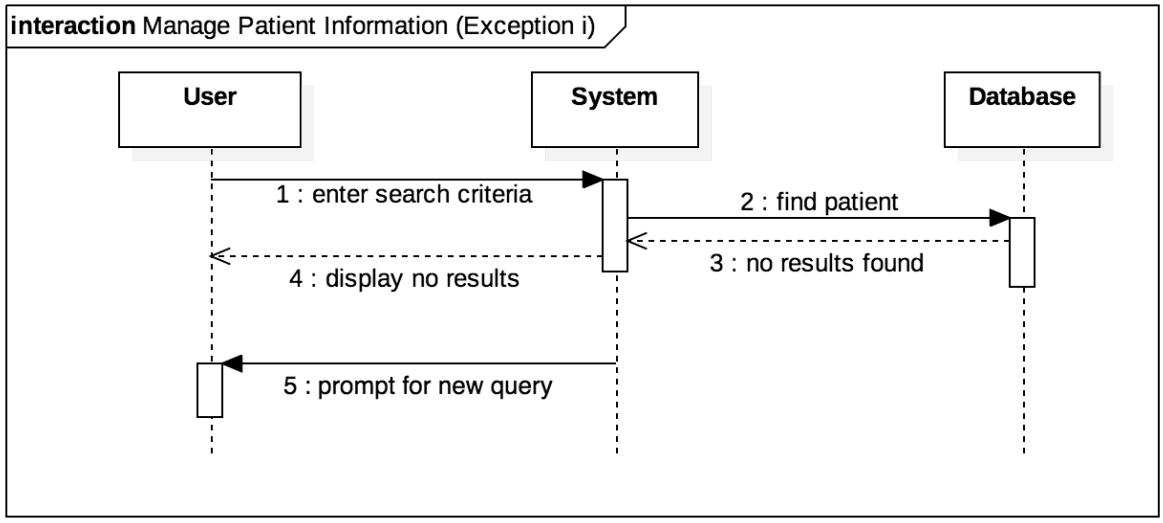

interaction Logout
«Actor»
User
«boundary»
Login view
«boundary»
Logout view
«entity»
session
2 : clear session details
1 : press logout button
«destroy»
3 : delete session
4 : display

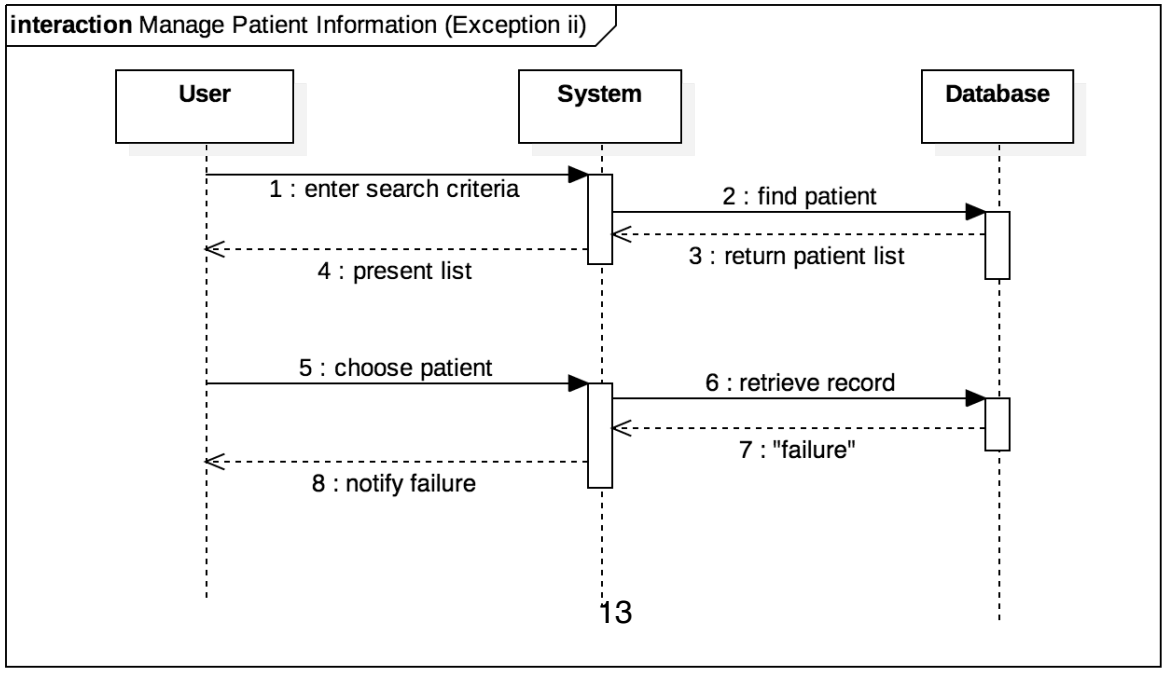
3. **Manage Patient Information**
    a. **Participants:** Administrative user
    b. **Precondition**
        i. The user who is trying to manage the patient information has the appropriate access.
        ii. The system has been logged into and is ready for interaction
    c. **Typical Course of Events**

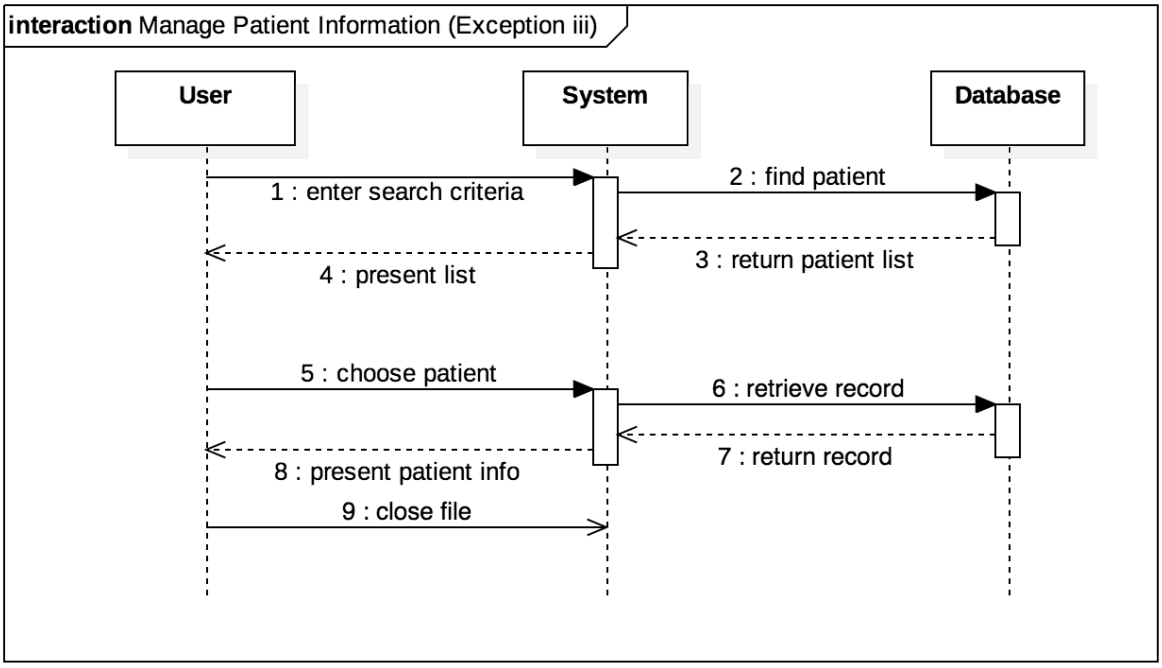
Actor Intentions	System Responsibilities
1. User enters criteria to find the patient whose information they want to manage	
	2. The system uses the criteria to present the user with a list of possible matching patients
3. User selects the appropriate patient	
	4. The system retrieves the patient's information
5. The user edits or adds the appropriate information then saves the patient record.	
	6. The system saves the patient's record and indicates a successful save to the user

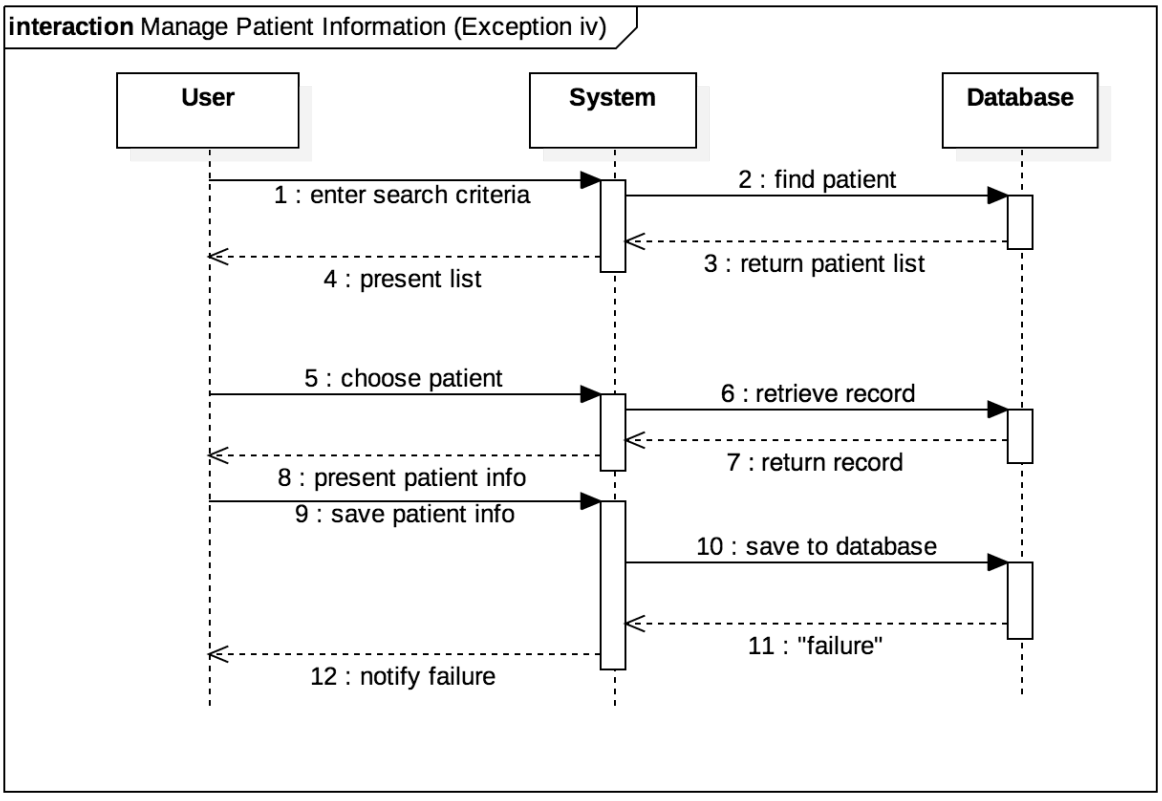
    d. **Alternative Courses**
        i. **Step 2:** The system was unable to find any matching patients based on the entered criteria, and prompts the user to try their search again
        ii. **Step 4:** The system is unable to retrieve the information from the database, and the user has to try again to select their patient
        iii. **Step 5:** After reviewing the information, the user realizes they don't need to make any changes, and cancel the editing action.
        iv. **Step 6:** The system is unable to save the information, and asks the user to try again.
    e. **Postcondition**
        i. The recently edited patient information is now available to anyone else who has access to that patient's record

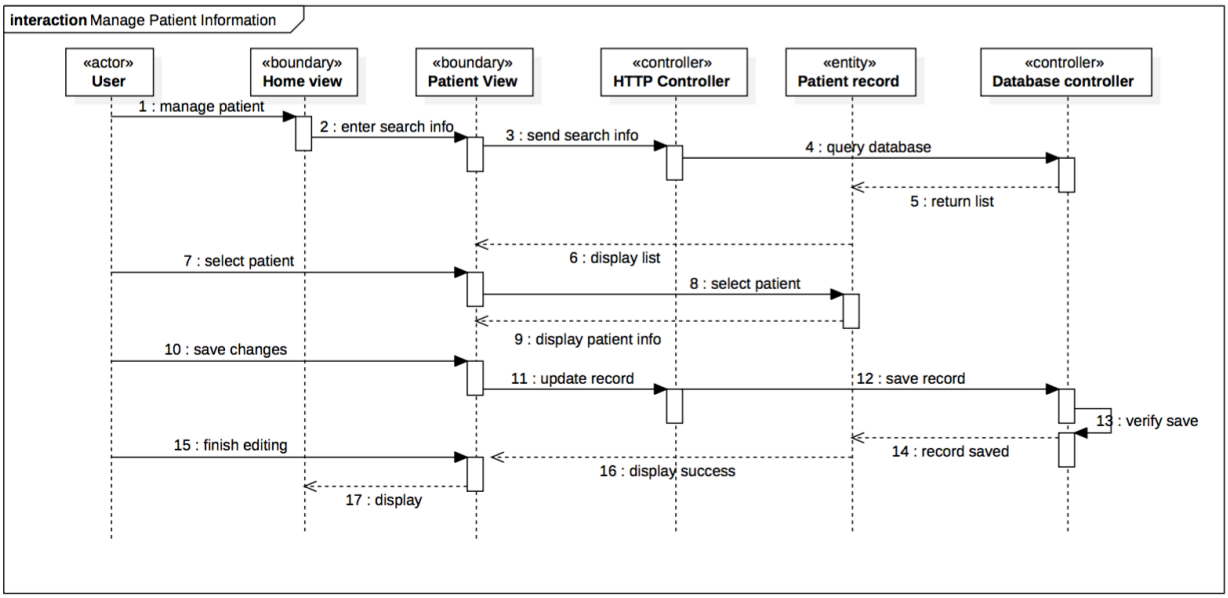

interaction Manage Patient Information (Actual)
User
System
Database
1 : enter search criteria
2 : find patient
3 : return patient list
4 : present list
5 : choose patient
6 : retrieve record
7 : return record
8 : present patient info
9 : save patient info
10 : save to database
11 : save successful
12 : save successful


interaction Manage Patient Information (Exception i)
User
System
Database
1 : enter search criteria
2 : find patient
3 : no results found
4 : display no results
5 : prompt for new query


interaction Manage Patient Information (Exception ii)
User
System
Database
1 : enter search criteria
2 : find patient
3 : return patient list
4 : present list
5 : choose patient
6 : retrieve record
7 : "failure"
8 : notify failure


interaction Manage Patient Information (Exception iii)
User
System
Database
1 : enter search criteria
2 : find patient
3 : return patient list
4 : present list
5 : choose patient
6 : retrieve record
7 : return record
8 : present patient info
9 : close file

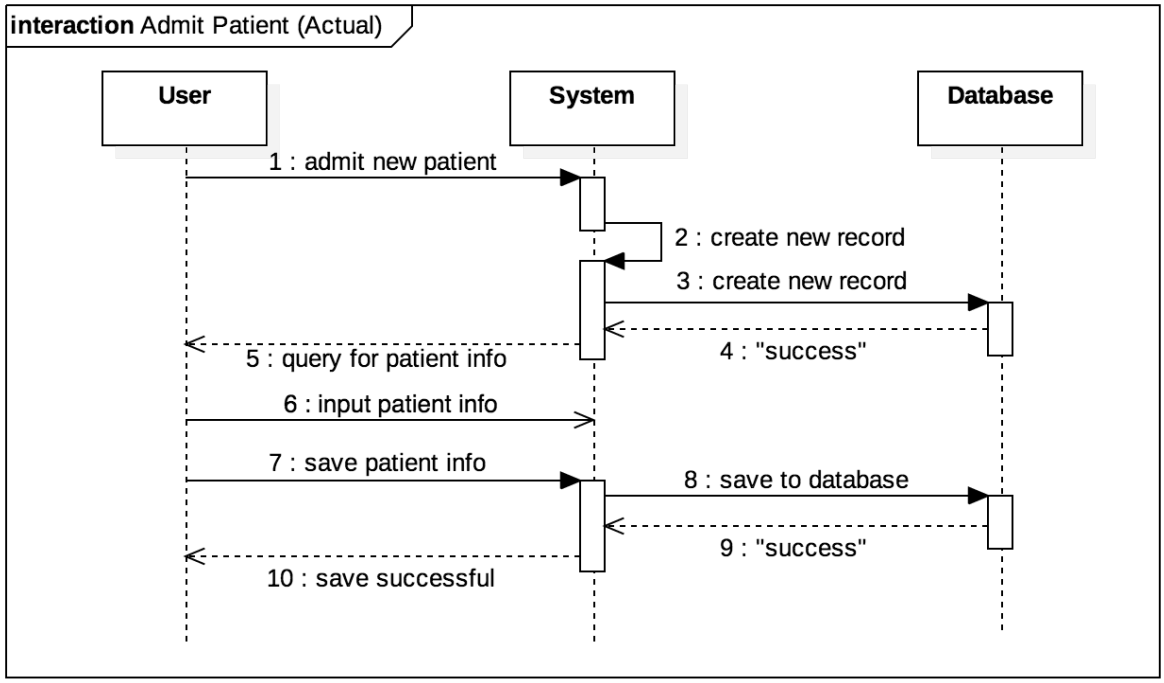

interaction Manage Patient Information (Exception iv)
User
System
Database
1 : enter search criteria
2 : find patient
3 : return patient list
4 : present list
5 : choose patient
6 : retrieve record
7 : return record
8 : present patient info
9 : save patient info
10 : save to database
11 : "failure"
12 : notify failure

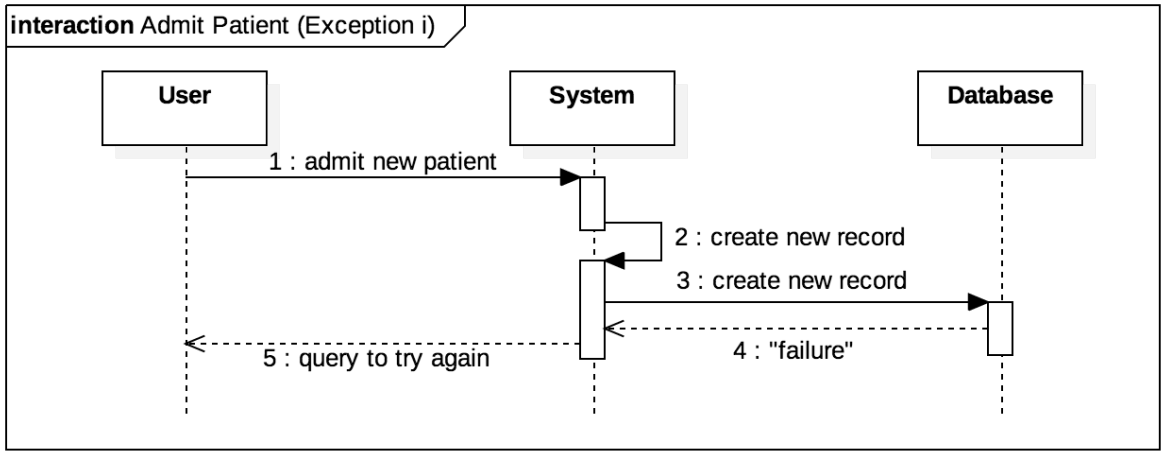

interaction Manage Patient Information
«actor»
User
«boundary»
Home view
«boundary»
Patient View
«controller»
HTTP Controller
«entity»
Patient record
«controller»
Database controller
1 : manage patient
2 : enter search info
3 : send search info
4 : query database
5 : return list
6 : display list
7 : select patient
8 : select patient
9 : display patient info
10 : save changes
11 : update record
12 : save record
13 : verify save
14 : record saved
15 : finish editing
16 : display success
17 : display

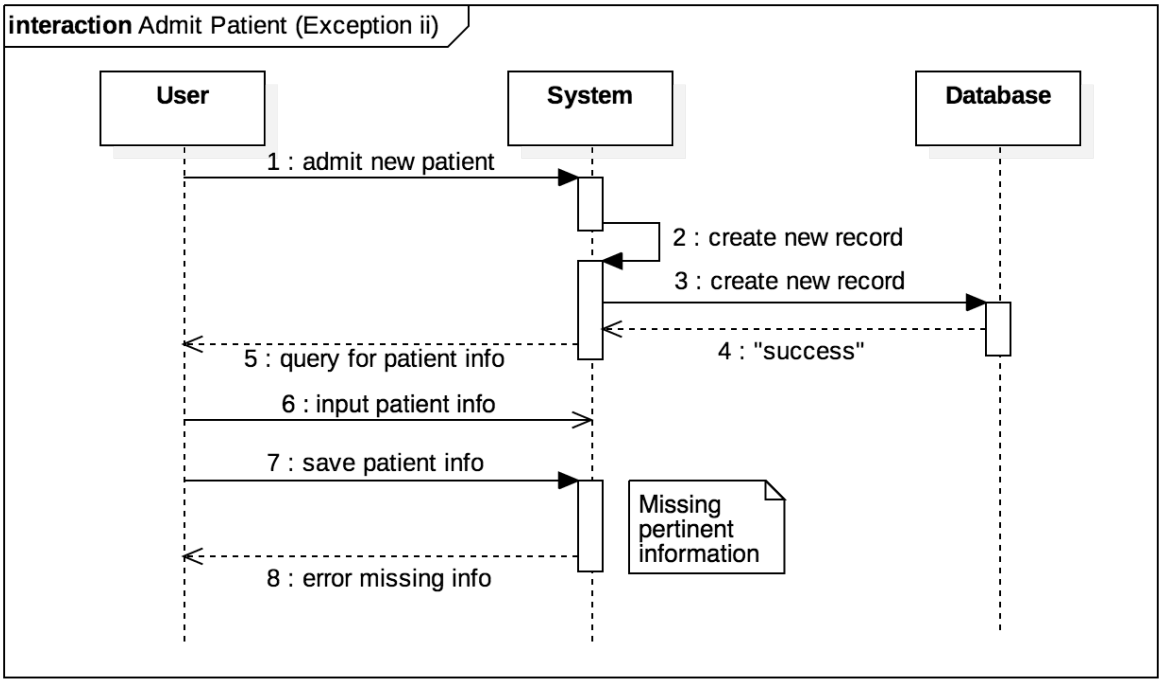
4. **Admit Patient**
    a. **Participants:** Administrative User
    b. **Precondition:** The user is logged into the system.
    c. **Typical Course of Events**

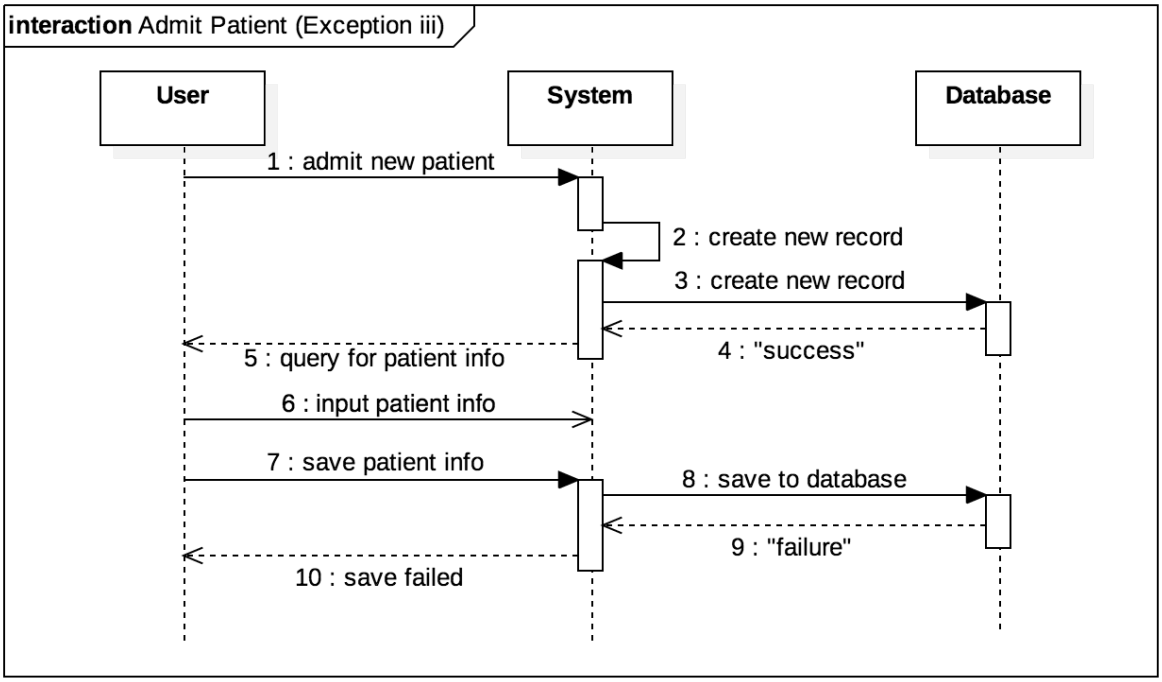
Actor Intentions	System Responsibilities
1. User indicates they want to admit a new patient	
	2. The system creates a new blank record for a patient, then asks the user to input admittance information
3. The user inputs the requested information, and saves the new patient record.	
	4. The system saves the new patient record and indicates a successful save to the user.

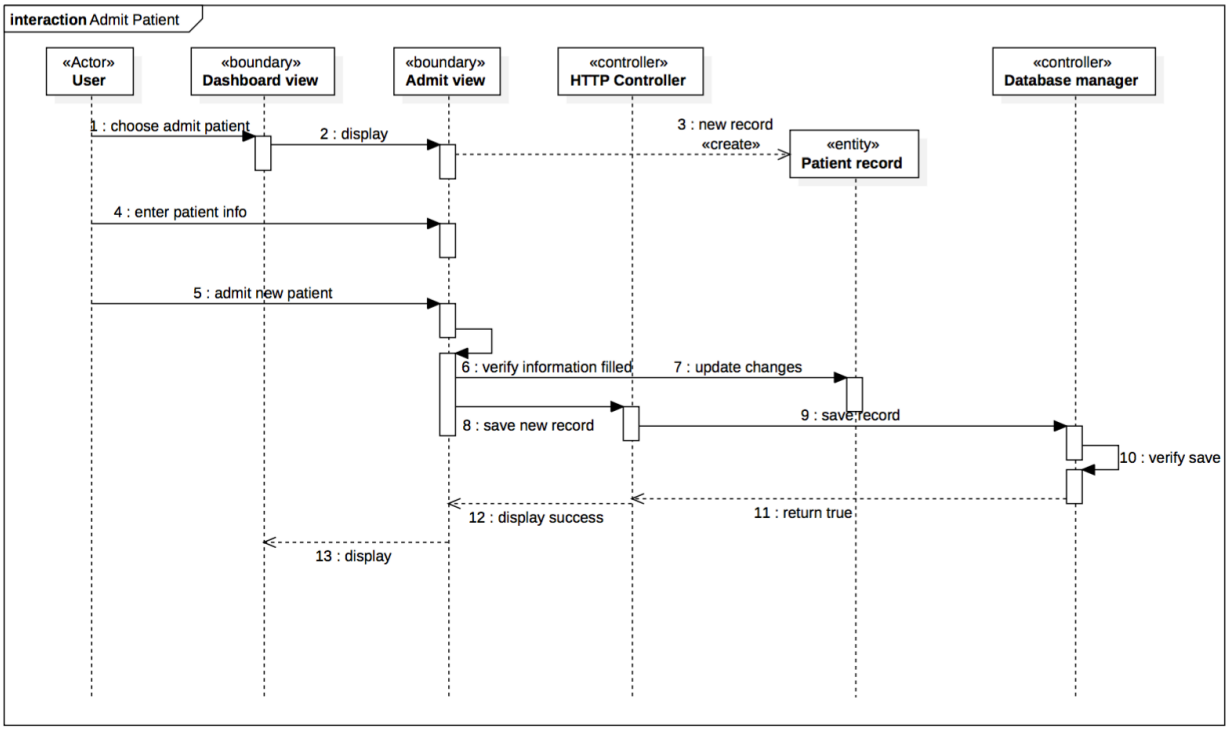
    d. **Alternative Courses**
        i. **Step 2:** The system is unable to create a new patient record, and asks the user to try again
        ii. **Step 3:** The user doesn't input enough correct information for the new record to be saved, so the system asks them to fill in the missing information
        iii. **Step 4:** The system isn't able to save the new record, and asks the user to try again
    e. **Postcondition**
        i. A new patient has been added to the patient database


interaction Admit Patient (Actual)
User
System
Database
1 : admit new patient
2 : create new record
3 : create new record
4 : "success"
5 : query for patient info
6 : input patient info
7 : save patient info
8 : save to database
9 : "success"
10 : save successful


interaction Admit Patient (Exception i)
User
System
Database
1 : admit new patient
2 : create new record
3 : create new record
4 : "failure"
5 : query to try again


interaction Admit Patient (Exception ii)
User
System
Database
1 : admit new patient
2 : create new record
3 : create new record
4 : "success"
5 : query for patient info
6 : input patient info
7 : save patient info
Missing pertinent information
8 : error missing info

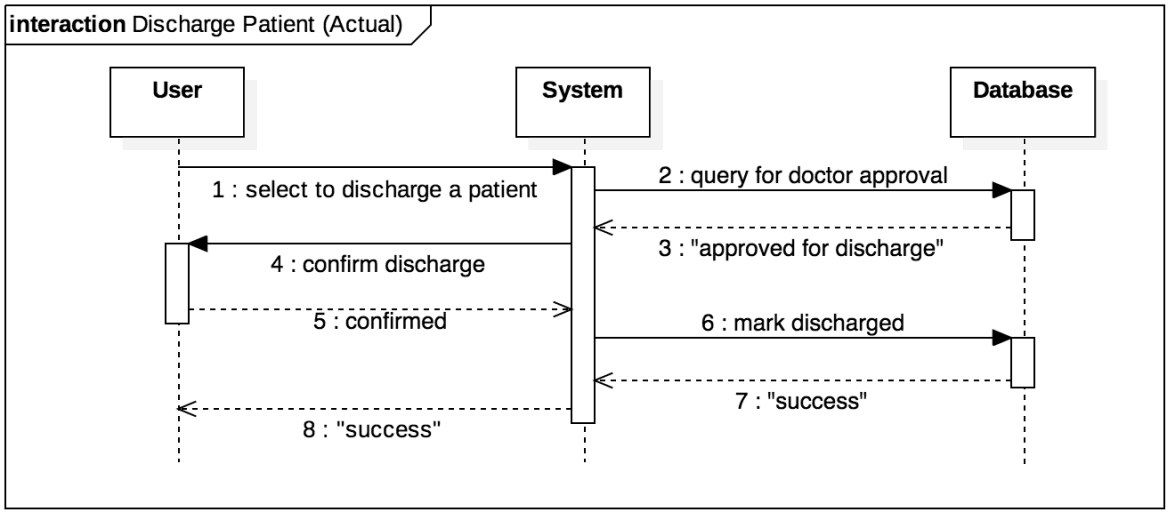

interaction Admit Patient (Exception iii)
User
System
Database
1 : admit new patient
2 : create new record
3 : create new record
4 : "success"
5 : query for patient info
6 : input patient info
7 : save patient info
8 : save to database
9 : "failure"
10 : save failed

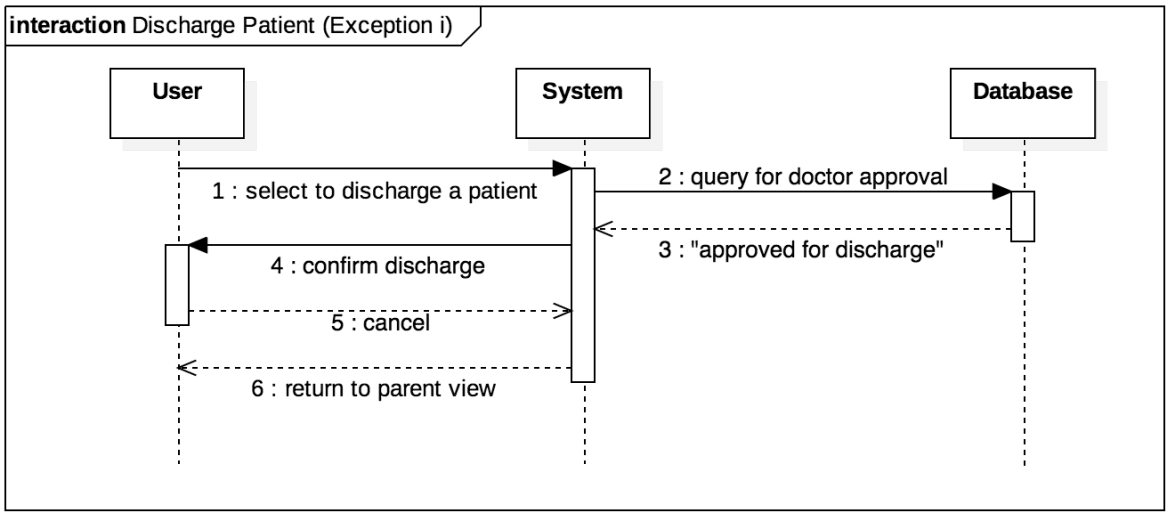

interaction Admit Patient
«Actor»
User
«boundary»
Dashboard view
«boundary»
Admit view
«controller»
HTTP Controller
«controller»
Database manager
«entity»
Patient record
1 : choose admit patient
2 : display
3 : new record
«create»
4 : enter patient info
5 : admit new patient
6 : verify information filled
7 : update changes
8 : save new record
9 : save record
10 : verify save
11 : return true
12 : display success
13 : display

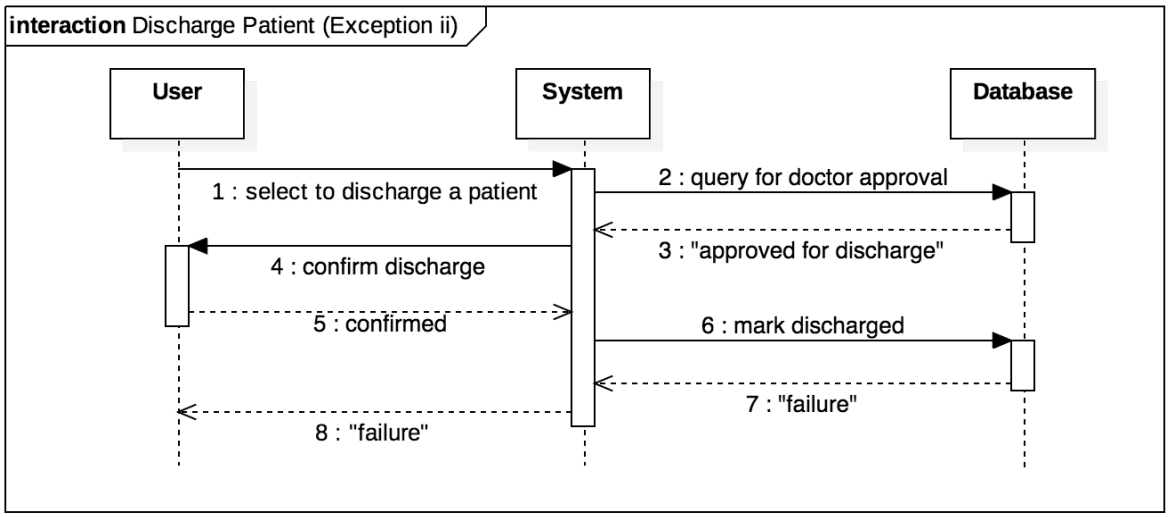
5. **Discharge Patient**
    a. **Participants:** Administrative user
    b. **Preconditions**
        i. The user has appropriate access to discharge a patient
        ii. The patient is still currently active in the system
        iii. The user is logged into the system
    c. **Typical Course of Events**

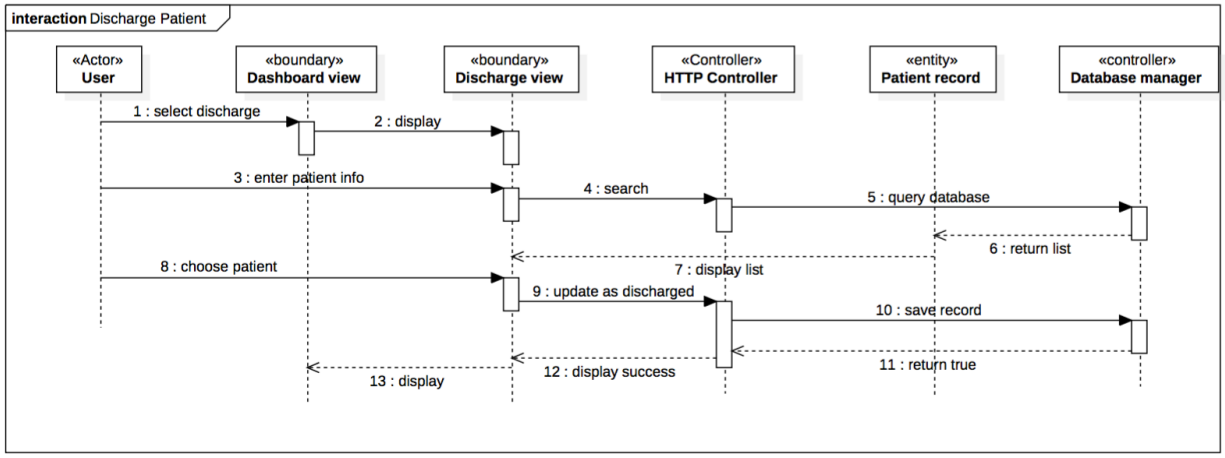
Actor Intentions	System Responsibilities
1. User indicates that they want to discharge a patient	
	2. The system confirms the users choice
3. The user confirms	
	4. The system verifies that the patient has been cleared for discharge by their doctor
	5. The system marks the patient as discharged

    d. **Alternative Courses**
        i. **Step 3:** The user decides they don't want to discharge the patient and cancels the interaction
        ii. **Step 4:** The patient has not been marked as ready to discharge, and the system alerts the user that they can't perform the requested action
    e. **Postcondition:**
        i. The patient has been marked as discharged, but their patient record remains in tact on the system database


interaction Discharge Patient (Actual)
User
System
Database
1 : select to discharge a patient
2 : query for doctor approval
3 : "approved for discharge"
4 : confirm discharge
5 : confirmed
6 : mark discharged
7 : "success"
8 : "success"


interaction Discharge Patient (Exception i)
User
System
Database
1 : select to discharge a patient
2 : query for doctor approval
3 : "approved for discharge"
4 : confirm discharge
5 : cancel
6 : return to parent view

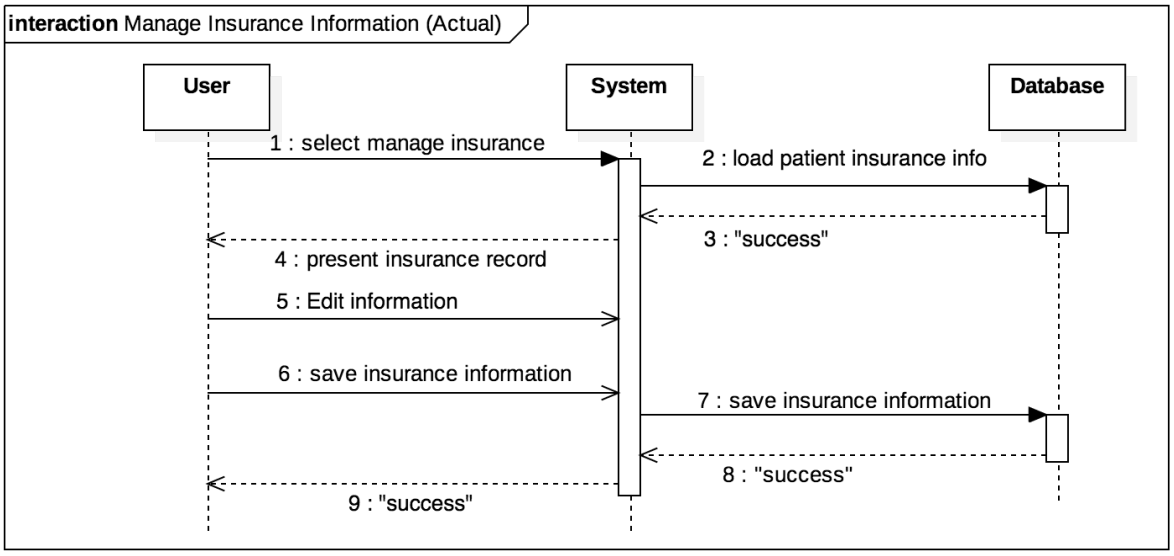

interaction Discharge Patient (Exception ii)
User
System
Database
1 : select to discharge a patient
2 : query for doctor approval
3 : "approved for discharge"
4 : confirm discharge
5 : confirmed
6 : mark discharged
7 : "failure"
8 : "failure"

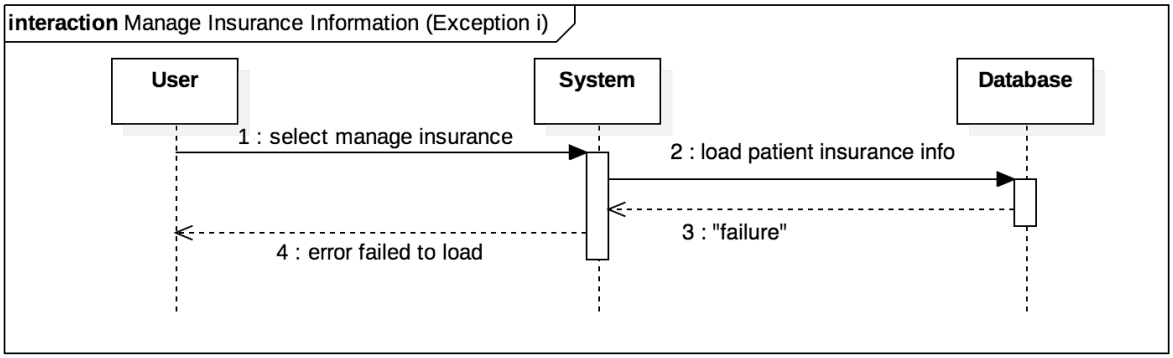

interaction Discharge Patient
«Actor»
User
«boundary»
Dashboard view
«boundary»
Discharge view
«Controller»
HTTP Controller
«entity»
Patient record
«controller»
Database manager
1 : select discharge
2 : display
3 : enter patient info
4 : search
5 : query database
6 : return list
7 : display list
8 : choose patient
9 : update as discharged
10 : save record
11 : return true
12 : display success
13 : display

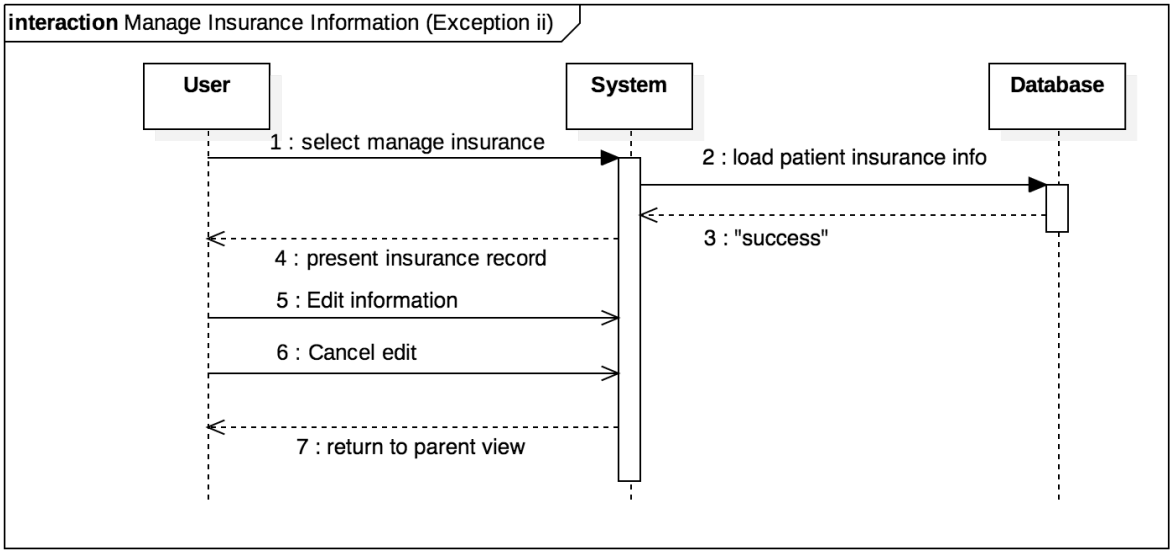
6. **Manage Patient Insurance Information**
    a. **Participants:** Administrative user
    b. **Preconditions**
        i. The user is logged into the system
    c. **Typical Course of Events**

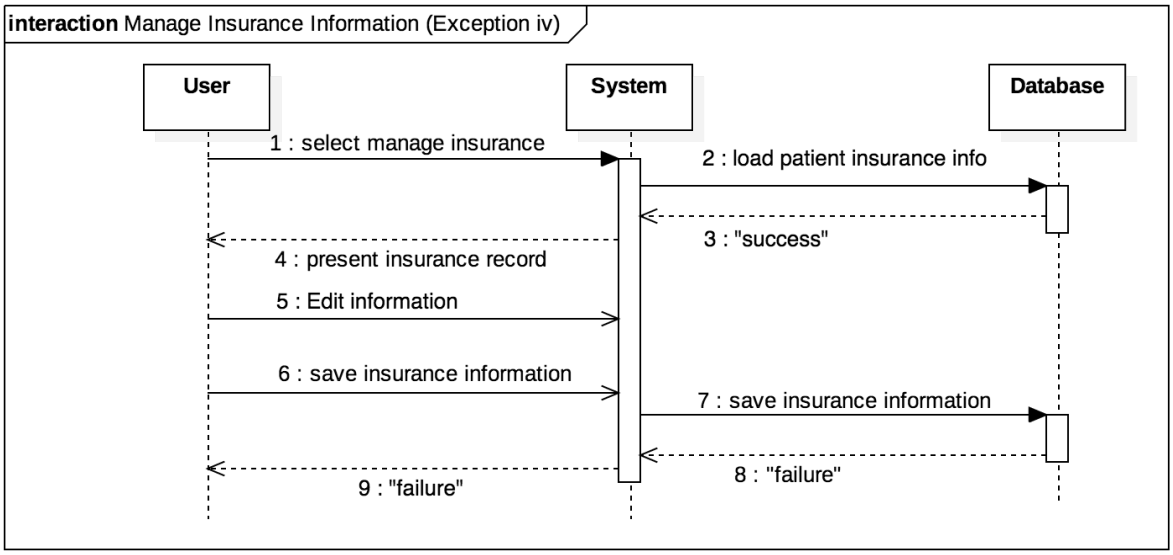
Actor Intentions	System Responsibilities
1. The user indicates they want to manage insurance information	
	2. The system presents the user with a new page that contains the patient's insurance information
3. The user modifies the insurance information	
4. The user indicates they want to save the new insurance information	
	5. The system saves the new insurance information to the patient's record

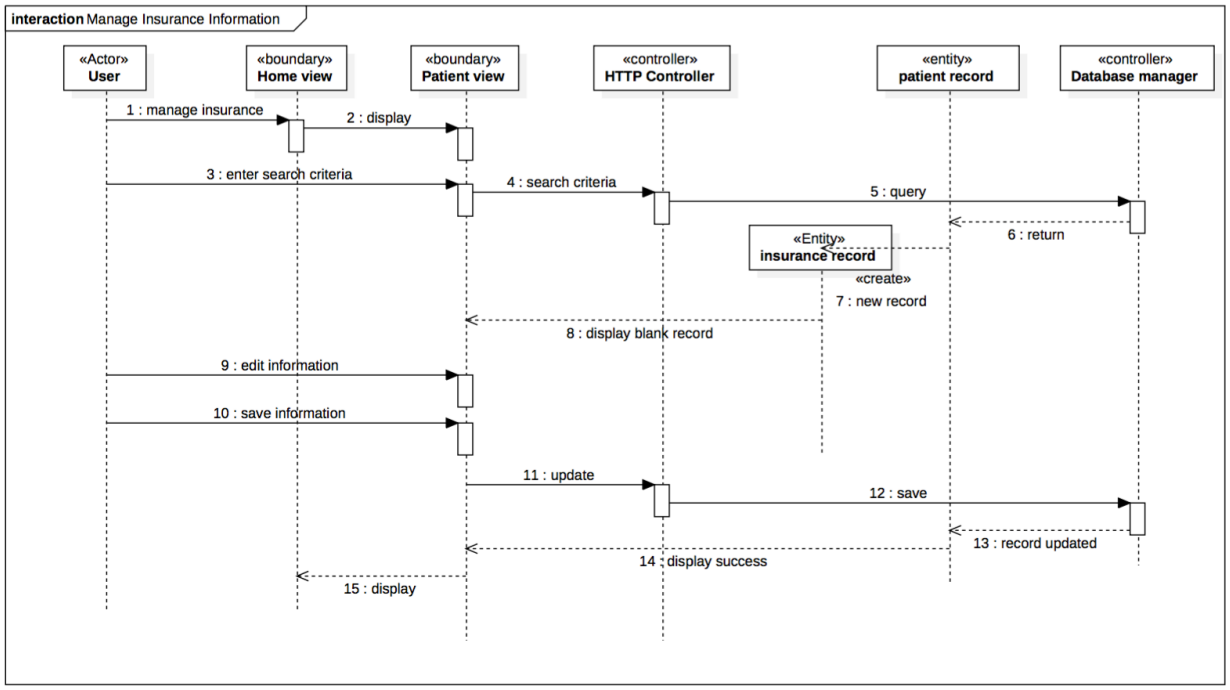
    d. **Alternative Courses**
        i. **Step 1:** The system is unable to load the corresponding insurance information and asks the user to try again
        ii. **Step 3:** The user has viewed the insurance information, and decides no changes need to be made, and cancels the interaction
        iv. **Step 5:** The system is unable to save the insurance information asks the user to try again
    e. **Postcondition**
        i. The patient's insurance information reflects the most recent information available.


interaction Manage Insurance Information (Actual)
User
System
Database
1 : select manage insurance
2 : load patient insurance info
3 : "success"
4 : present insurance record
5 : Edit information
6 : save insurance information
7 : save insurance information
8 : "success"
9 : "success"


interaction Manage Insurance Information (Exception i)
User
System
Database
1 : select manage insurance
2 : load patient insurance info
3 : "failure"
4 : error failed to load


interaction Manage Insurance Information (Exception ii)
User
System
Database
1 : select manage insurance
2 : load patient insurance info
3 : "success"
4 : present insurance record
5 : Edit information
6 : Cancel edit
7 : return to parent view

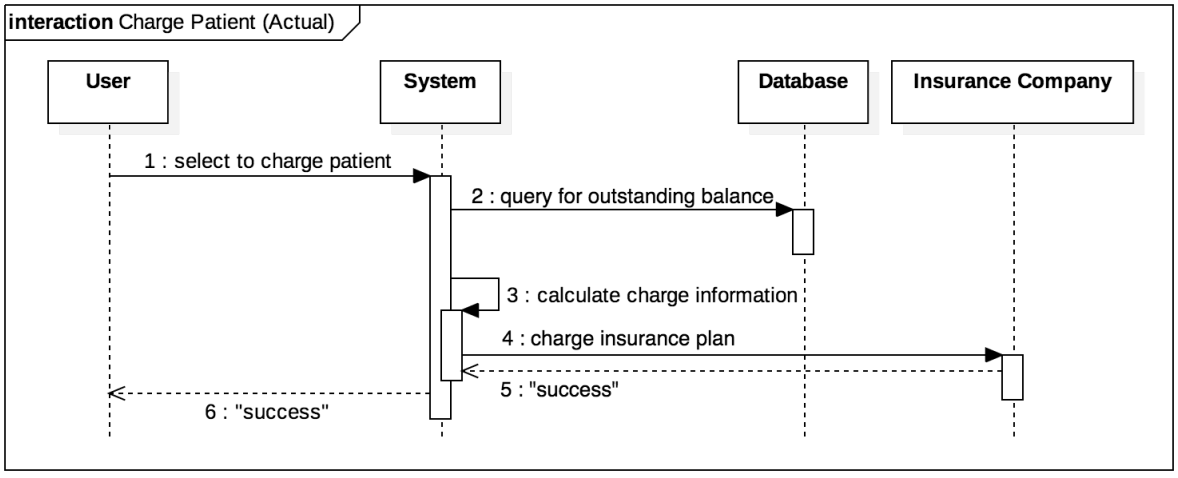

interaction Manage Insurance Information (Exception iv)
User
System
Database
1 : select manage insurance
2 : load patient insurance info
3 : "success"
4 : present insurance record
5 : Edit information
6 : save insurance information
7 : save insurance information
8 : "failure"
9 : "failure"

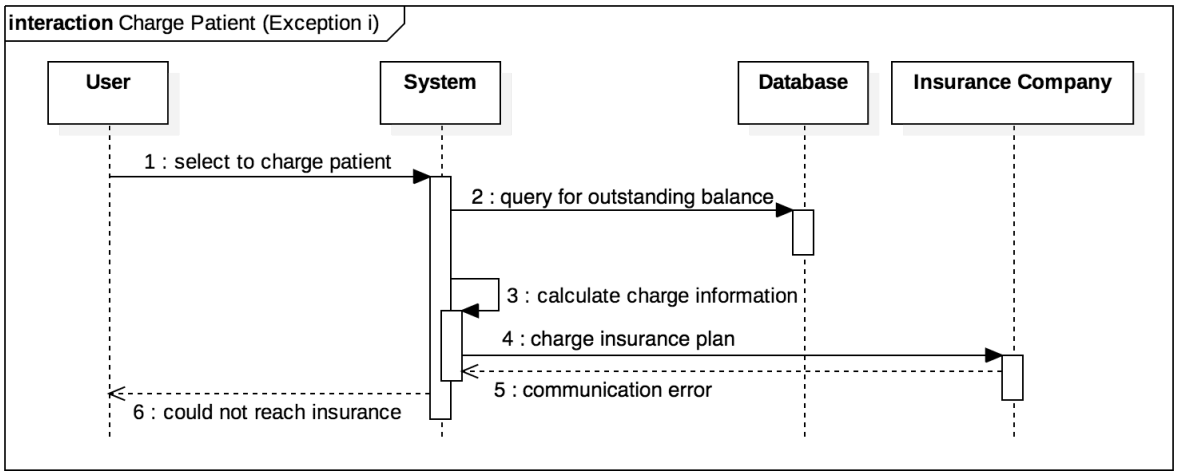

interaction Manage Insurance Information
«Actor» User
«boundary» Home view
«boundary» Patient view
«controller» HTTP Controller
«entity» patient record
«controller» Database manager
1 : manage insurance
2 : display
3 : enter search criteria
4 : search criteria
5 : query
6 : return
«Entity» insurance record
«create»
7 : new record
8 : display blank record
9 : edit information
10 : save information
11 : update
12 : save
13 : record updated
14 : display success
15 : display

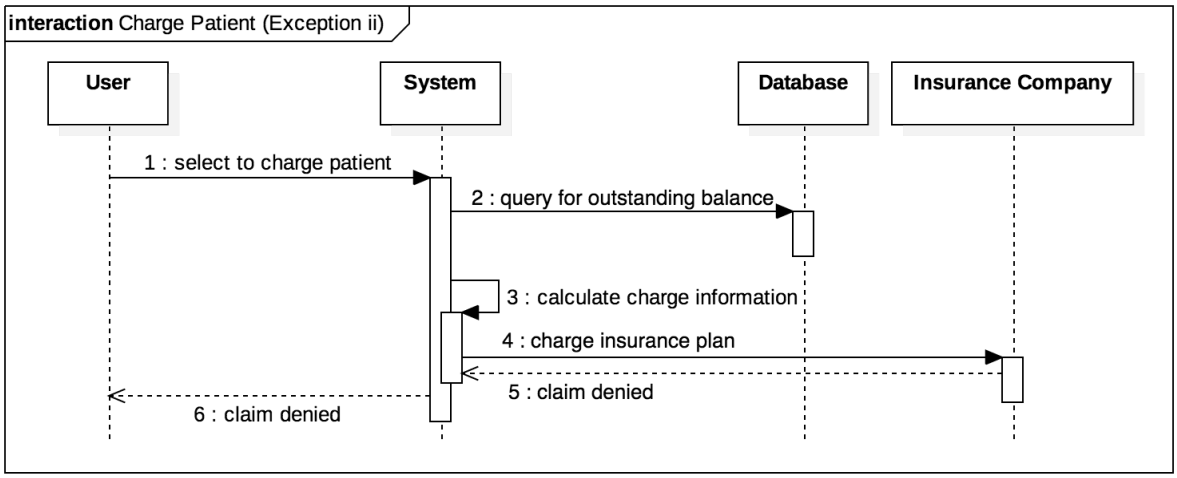
7. **Charge Patient**
    a. **Participants:** Administrative user
    b. **Preconditions**
        i. The patient has a file
        ii. The patient has been marked as able to be discharged
    c. **Typical Course of Events**

Actor Intentions	System Responsibilities
1. User indicates they would like to checkout a patient	
	2. The system calculates the total price based on services rendered and treatments prescribed
	3. The system saves the charge information
	4. The system sends the information to the insurance company for processing

    d. **Alternative Courses**
        i. **Step 4:** The system is unable to communicate with the insurance company at this time, and asks the user to try again later
        ii. **Step 4:** The patient's insurance company has denied the patient's claim, and alerts the user of the problem
    e. **Postcondition**
        i. The patient has been processed and charged



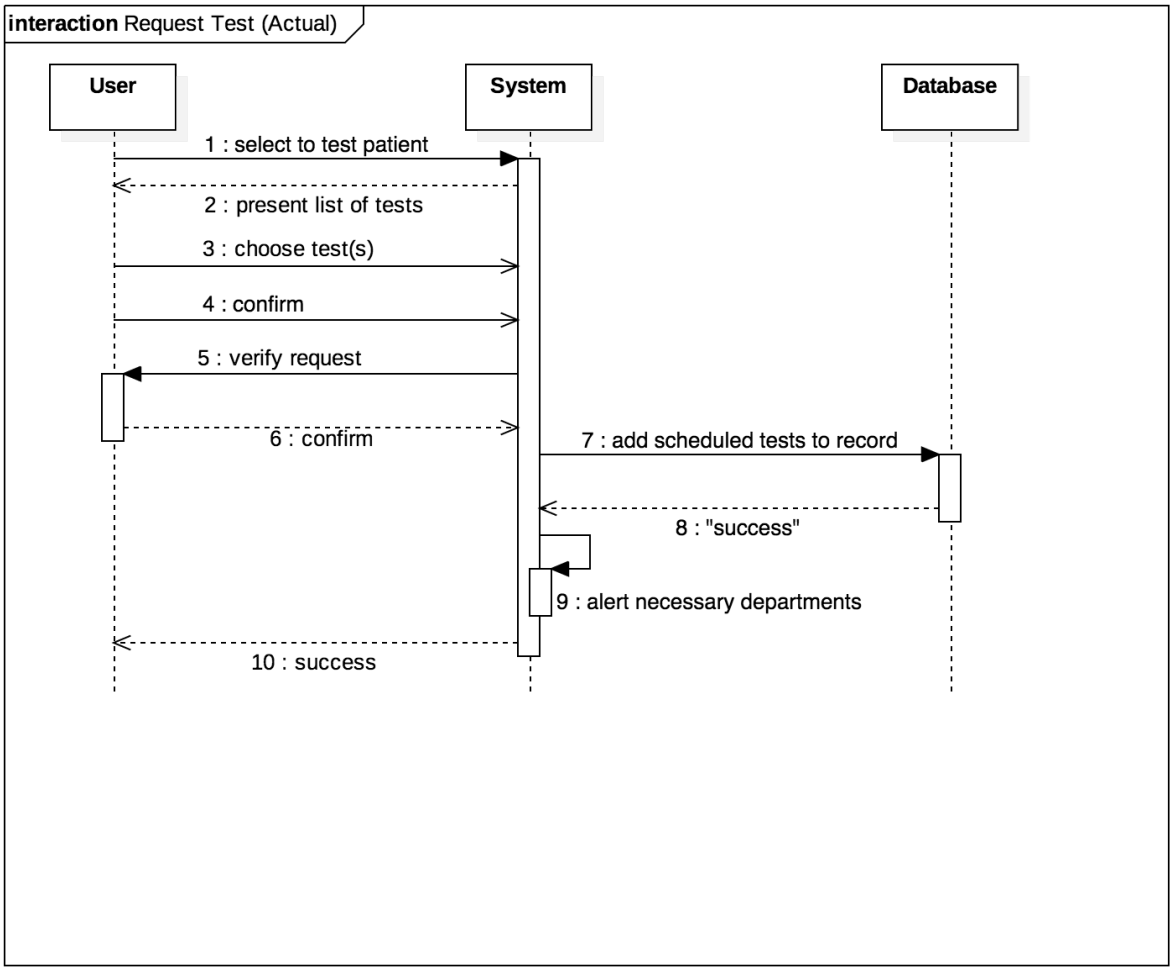

interaction Charge Patient (Exception i)
User
System
Database
Insurance Company
1 : select to charge patient
2 : query for outstanding balance
3 : calculate charge information
4 : charge insurance plan
5 : communication error
6 : could not reach insurance

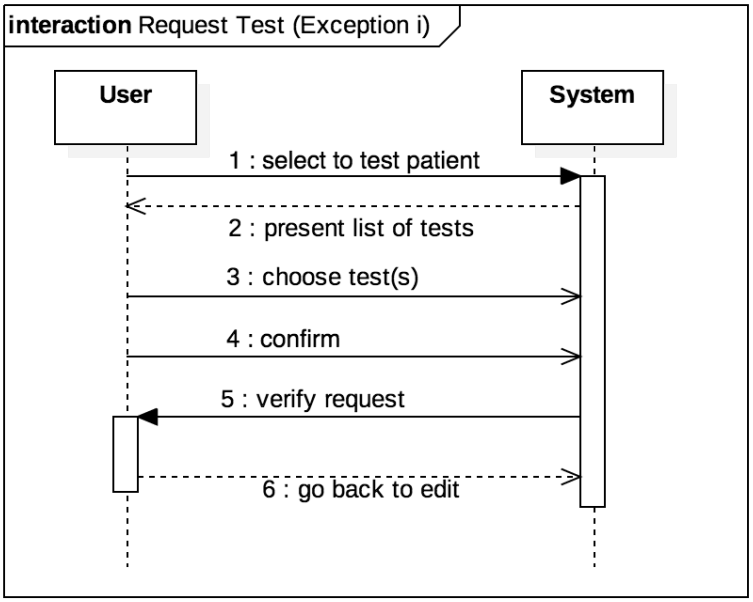

interaction Charge Patient (Exception ii)
User
System
Database
Insurance Company
1 : select to charge patient
2 : query for outstanding balance
3 : calculate charge information
4 : charge insurance plan
5 : claim denied
6 : claim denied

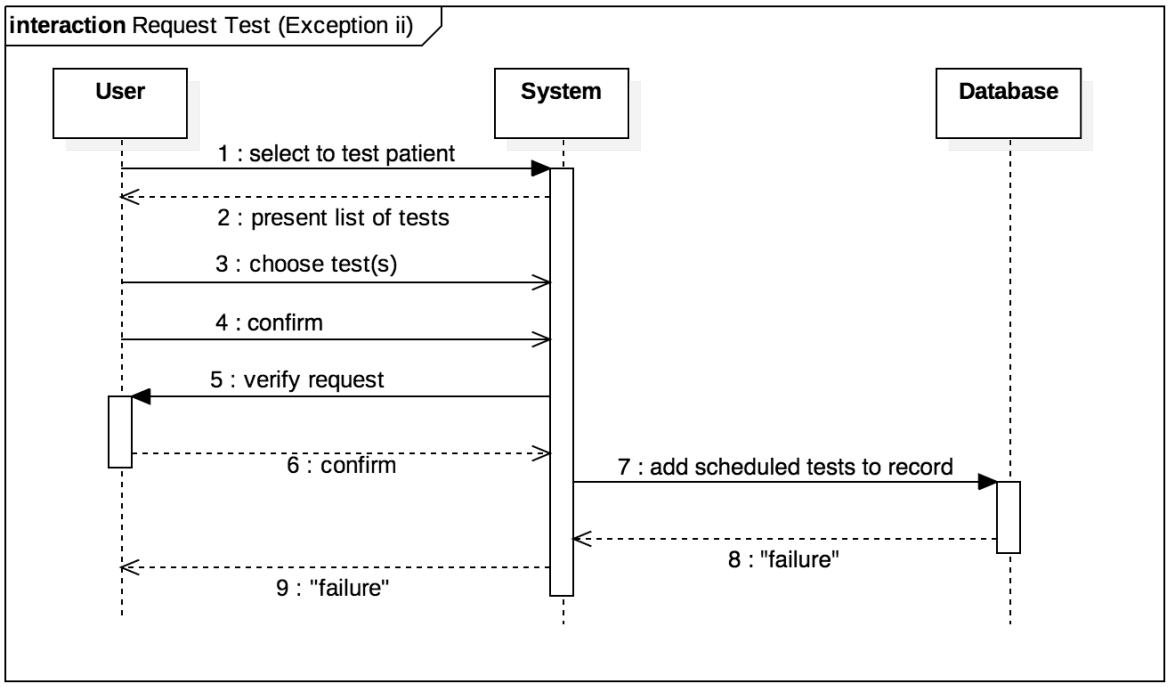
8. **Request Test**
    a. **Participants:** Technical user, Operational user
    b. **Preconditions**
        i. The requesting user is logged in
        ii. The requesting user has access to a patient file
    c. **Typical Course of Events**

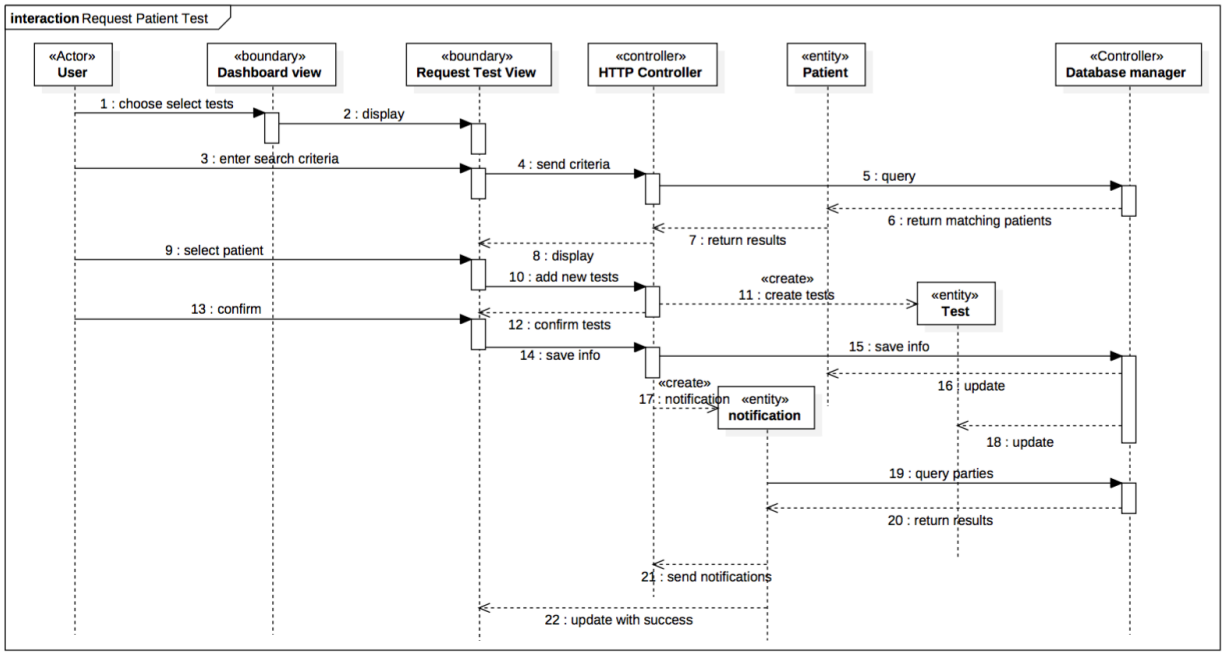
Actor Intentions	System Responsibilities
1. User selects to request a test on a patient	
	2. The system presents the user with a list of tests that can be scheduled
3. The user selects which test(s) to schedule	
	4. The system asks the user to verify the scheduled test(s)
5. The user verifies the test(s) they want scheduled	
	6. The system schedules the patient for the requested test(s) and alerts the appropriate departments

    d. **Alternative Courses**
        i. **Step 5:** The user realizes that changes need to be made to the requested tests, and needs to go back to change them
        ii. **Step 6:** The system isn't able to save the information and asks the user to try again
    e. **Postconditions**
        i. The patient has been scheduled for any tests that they needed


interaction Request Test (Actual)
User
System
Database
1 : select to test patient
2 : present list of tests
3 : choose test(s)
4 : confirm
5 : verify request
6 : confirm
7 : add scheduled tests to record
8 : "success"
9 : alert necessary departments
10 : success


interaction Request Test (Exception i)
User
System
1 : select to test patient
2 : present list of tests
3 : choose test(s)
4 : confirm
5 : verify request
6 : go back to edit

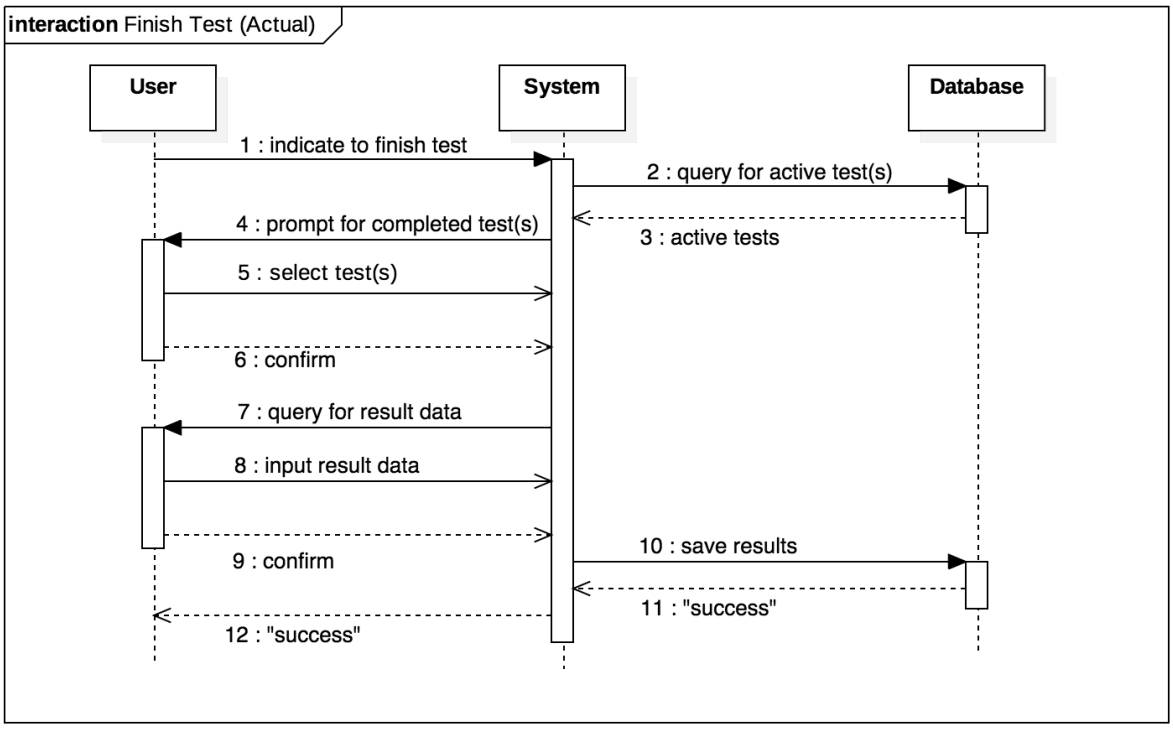

interaction Request Test (Exception ii)
User
System
Database
1 : select to test patient
2 : present list of tests
3 : choose test(s)
4 : confirm
5 : verify request
6 : confirm
7 : add scheduled tests to record
8 : "failure"
9 : "failure"

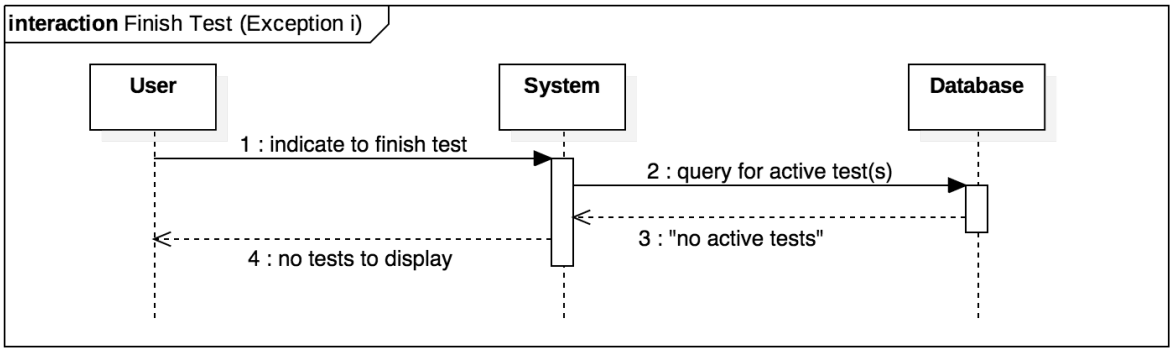

interaction Request Patient Test
«Actor» User
«boundary» Dashboard view
«boundary» Request Test View
«controller» HTTP Controller
«entity» Patient
«Controller» Database manager
1 : choose select tests
2 : display
3 : enter search criteria
4 : send criteria
5 : query
6 : return matching patients
7 : return results
8 : display
9 : select patient
10 : add new tests
«create»
11 : create tests
«entity» Test
12 : confirm tests
13 : confirm
14 : save info
15 : save info
16 : update
«create»
17 : notification
«entity» notification
18 : update
19 : query parties
20 : return results
21 : send notifications
22 : update with success

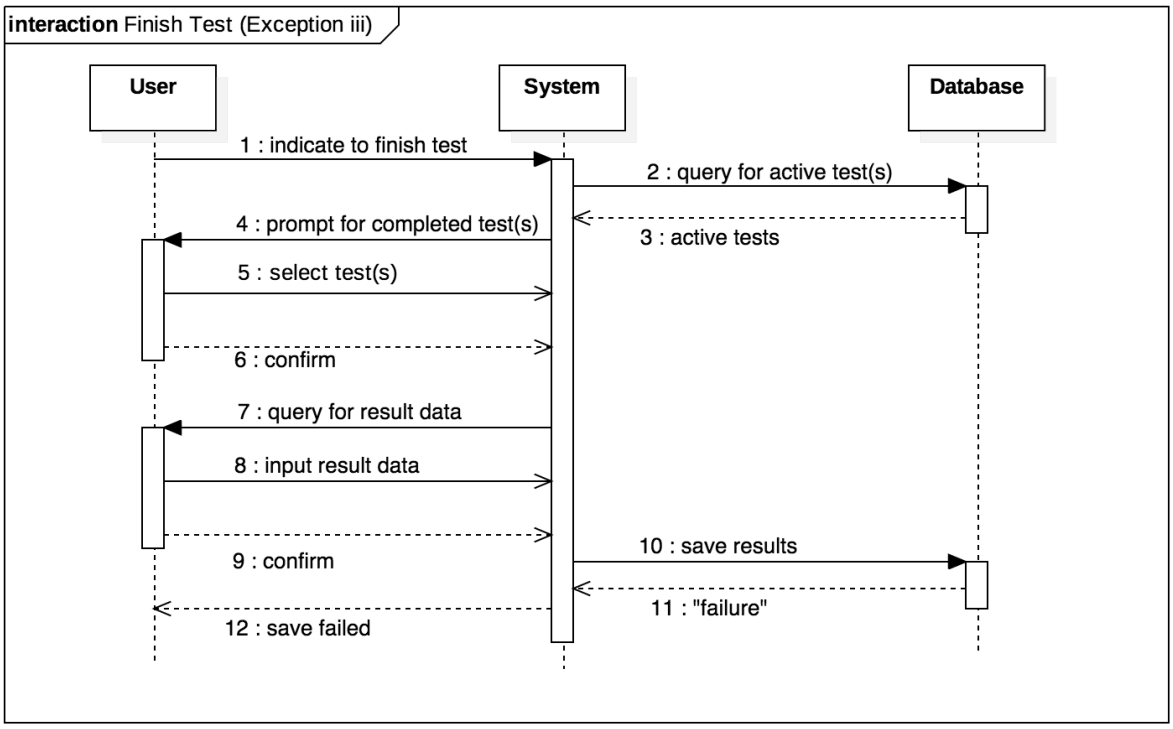
9. **Finish Test**
    a. **Participants:** Technical user, Operational user
    b. **Preconditions**
        i. The technical user is viewing a patient record
    c. **Typical Course of Events**

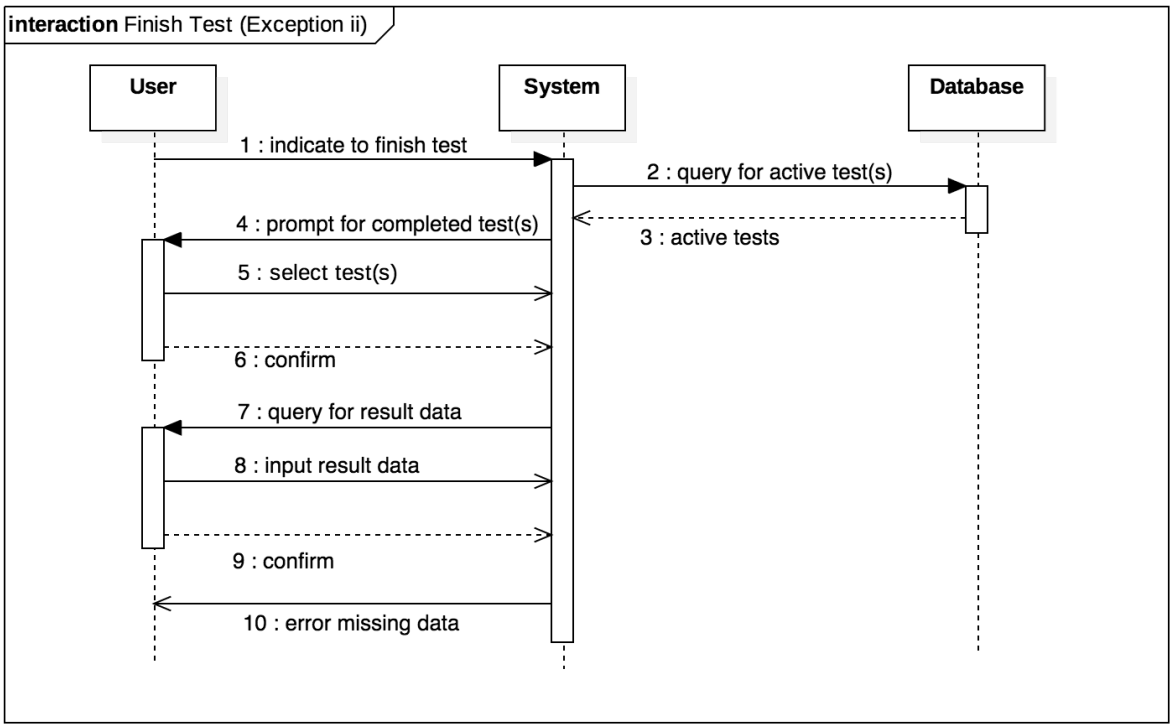
Actor Intentions	System Responsibilities
1. The user indicates they want to submit test results	
	2. The system displays a list of active tests for the patient
3. The user selects which tests are finished	
	4. The system prompts the user to fill in the test results for each test that has been completed
5. The user enters all of the test results and saves them to the system	
	6. The system saves the results of the test and alerts the primary care doctor of the new results

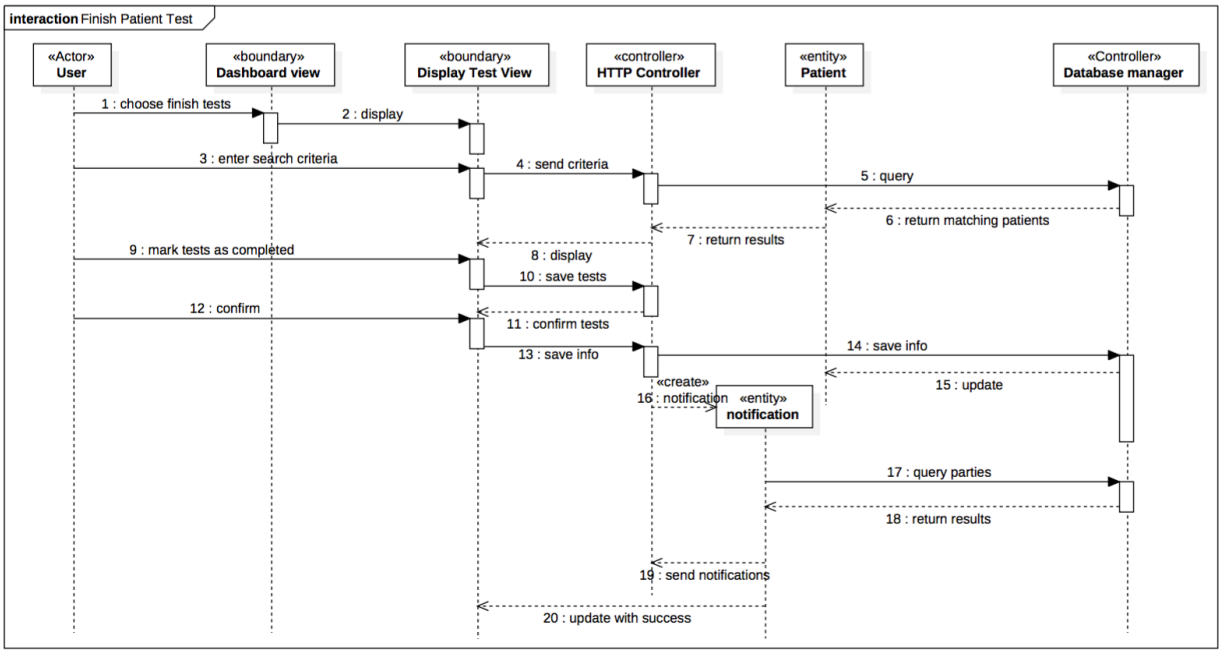
    d. **Alternative Courses**
        i. **Step 2:** No tests to display
        ii. **Step 5:** The user omits pertinent data; the system alerts them of an incomplete form
        iii. **Step 6:** The system is unable to save the results and alerts the user to try again
    e. **Postcondition:** The tests that were selected as completed have now been updated in the system and are visible to other users


interaction Finish Test (Actual)
User
System
Database
1 : indicate to finish test
2 : query for active test(s)
3 : active tests
4 : prompt for completed test(s)
5 : select test(s)
6 : confirm
7 : query for result data
8 : input result data
9 : confirm
10 : save results
11 : "success"
12 : "success"


interaction Finish Test (Exception i)
User
System
Database
1 : indicate to finish test
2 : query for active test(s)
3 : "no active tests"
4 : no tests to display


interaction Finish Test (Exception iii)
User
System
Database
1 : indicate to finish test
2 : query for active test(s)
3 : active tests
4 : prompt for completed test(s)
5 : select test(s)
6 : confirm
7 : query for result data
8 : input result data
9 : confirm
10 : save results
11 : "failure"
12 : save failed

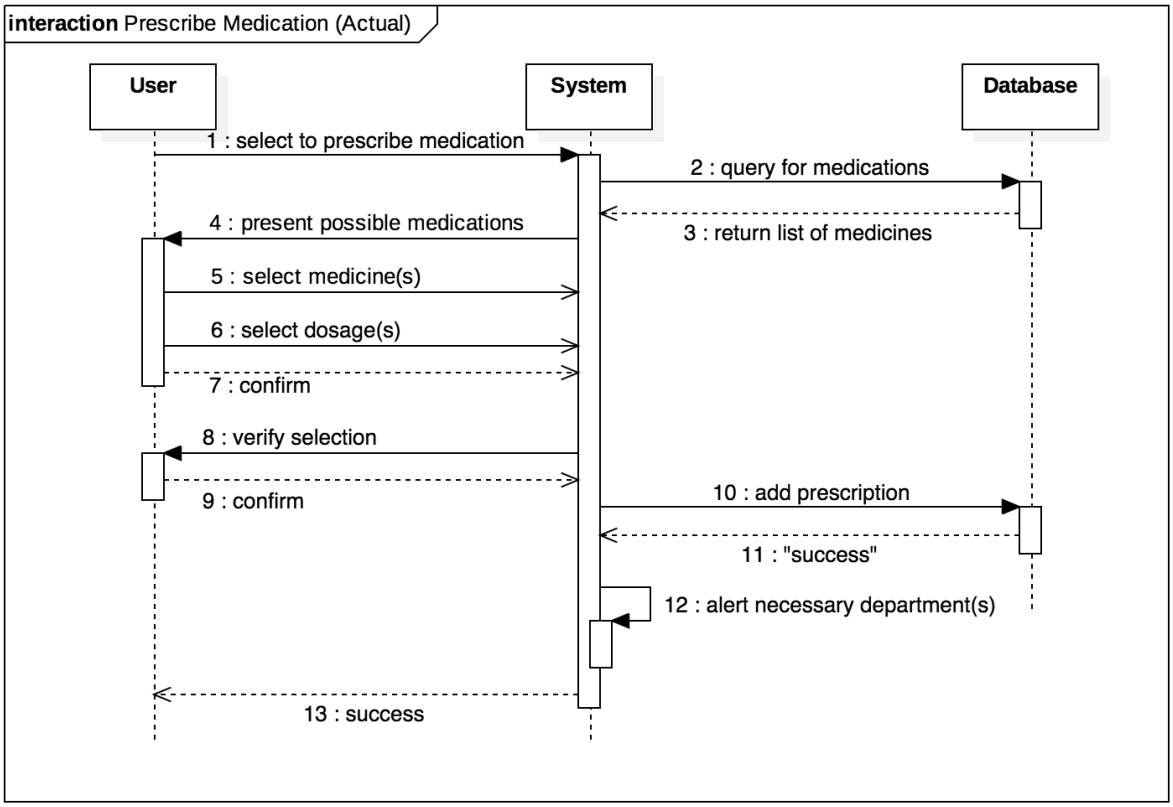

interaction Finish Test (Exception ii)
User
System
Database
1 : indicate to finish test
2 : query for active test(s)
3 : active tests
4 : prompt for completed test(s)
5 : select test(s)
6 : confirm
7 : query for result data
8 : input result data
9 : confirm
10 : error missing data

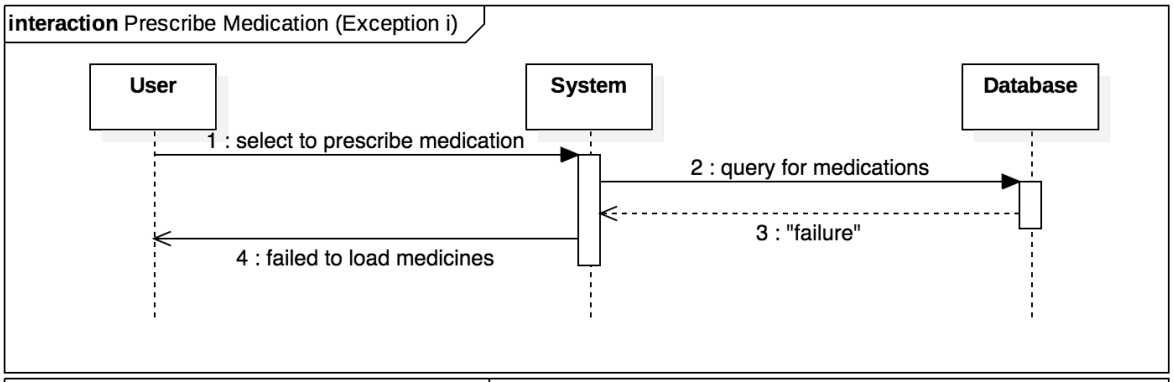

interaction Finish Patient Test
«Actor»
User
«boundary»
Dashboard view
«boundary»
Display Test View
«controller»
HTTP Controller
«entity»
Patient
«Controller»
Database manager
1 : choose finish tests
2 : display
3 : enter search criteria
4 : send criteria
5 : query
6 : return matching patients
7 : return results
8 : display
9 : mark tests as completed
10 : save tests
11 : confirm tests
12 : confirm
13 : save info
14 : save info
15 : update
«create»
16 : notification
«entity»
notification
17 : query parties
18 : return results
19 : send notifications
20 : update with success

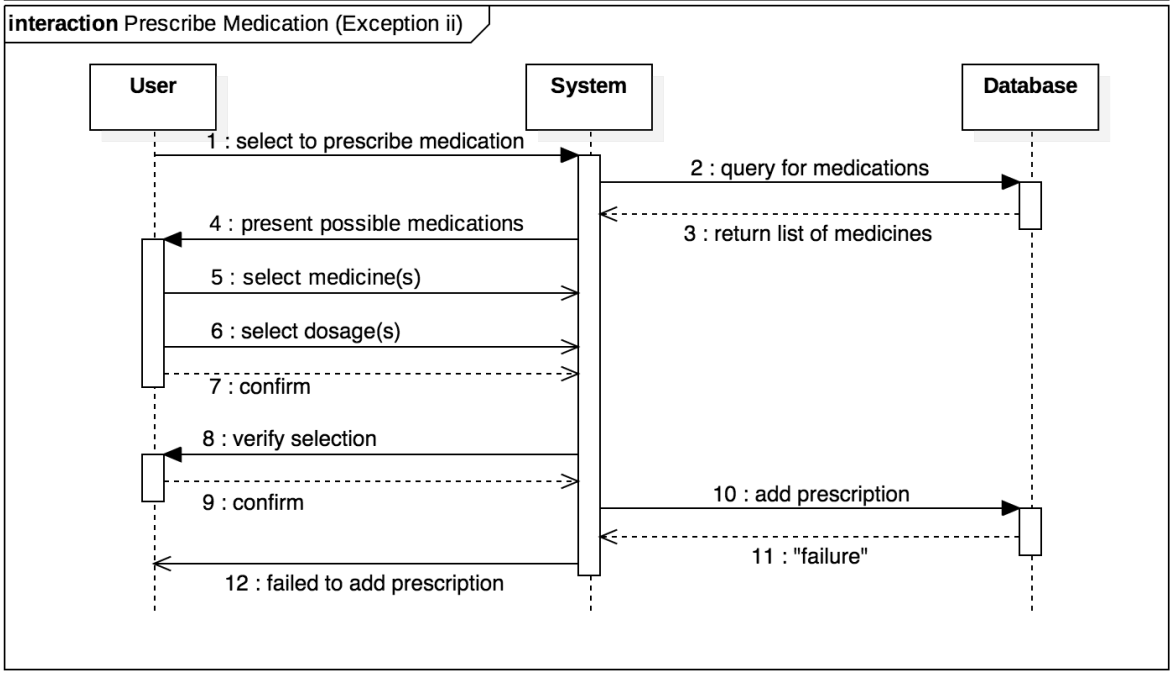
10. **Prescribe Medication**
    a. **Participants:** Operational user
    b. **Preconditions**
        i. The user is logged in
        ii. The user is viewing a patient record
        iii. The patient record has no conflicting allergies with the medication
    c. **Typical Course of Events**

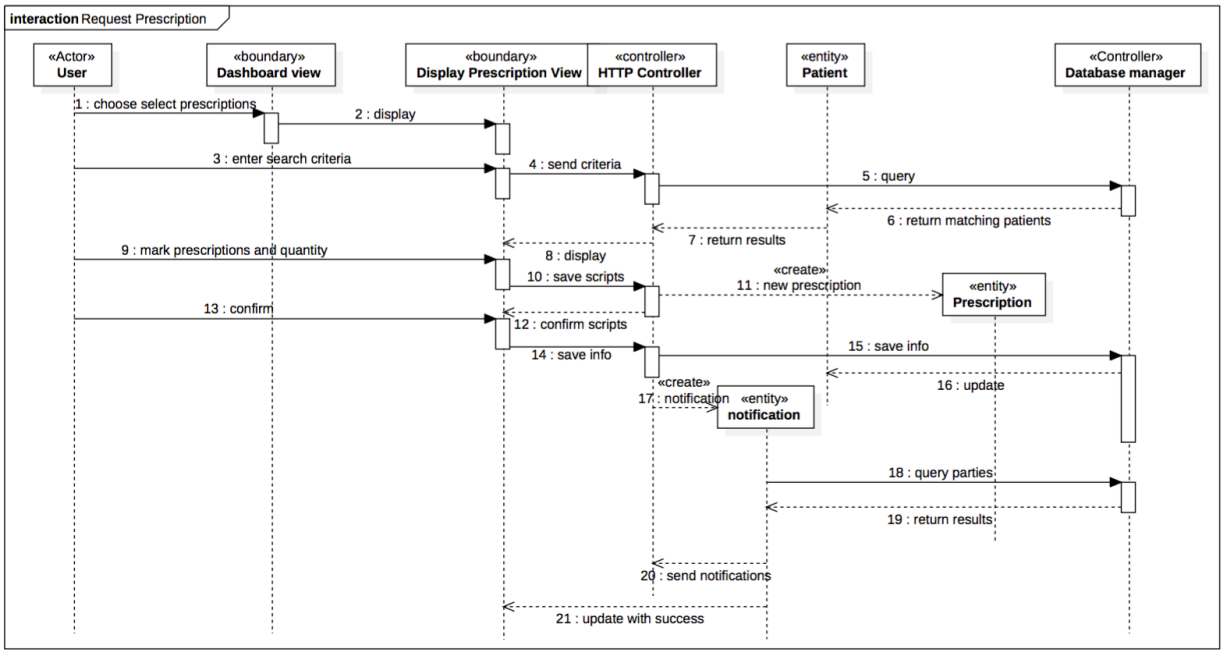
Actor Intentions	System Responsibilities
1. User selects to prescribe a medication	
	2. The system compiles a list of medicines based on the patients allergies and existing prescriptions
	3. The system presents the user with a list of available medications to prescribe
4. The user selects the medicine(s) and corresponding dosages they want to prescribe	
	5. The system confirms the selection
	6. The system saves the new prescription and transfers the prescription to the pharmacy

    d. **Alternative Courses**
        i. **Step 2:** The system was unable to load medicines
        ii. **Step 5:** The system was unable to save the new prescription
    e. **Postcondition:** The patient has been issued new prescriptions based on what the user has input into the system


interaction Prescribe Medication (Actual)
User
System
Database
1 : select to prescribe medication
2 : query for medications
3 : return list of medicines
4 : present possible medications
5 : select medicine(s)
6 : select dosage(s)
7 : confirm
8 : verify selection
9 : confirm
10 : add prescription
11 : "success"
12 : alert necessary department(s)
13 : success


interaction Prescribe Medication (Exception i)
User
System
Database
1 : select to prescribe medication
2 : query for medications
3 : "failure"
4 : failed to load medicines

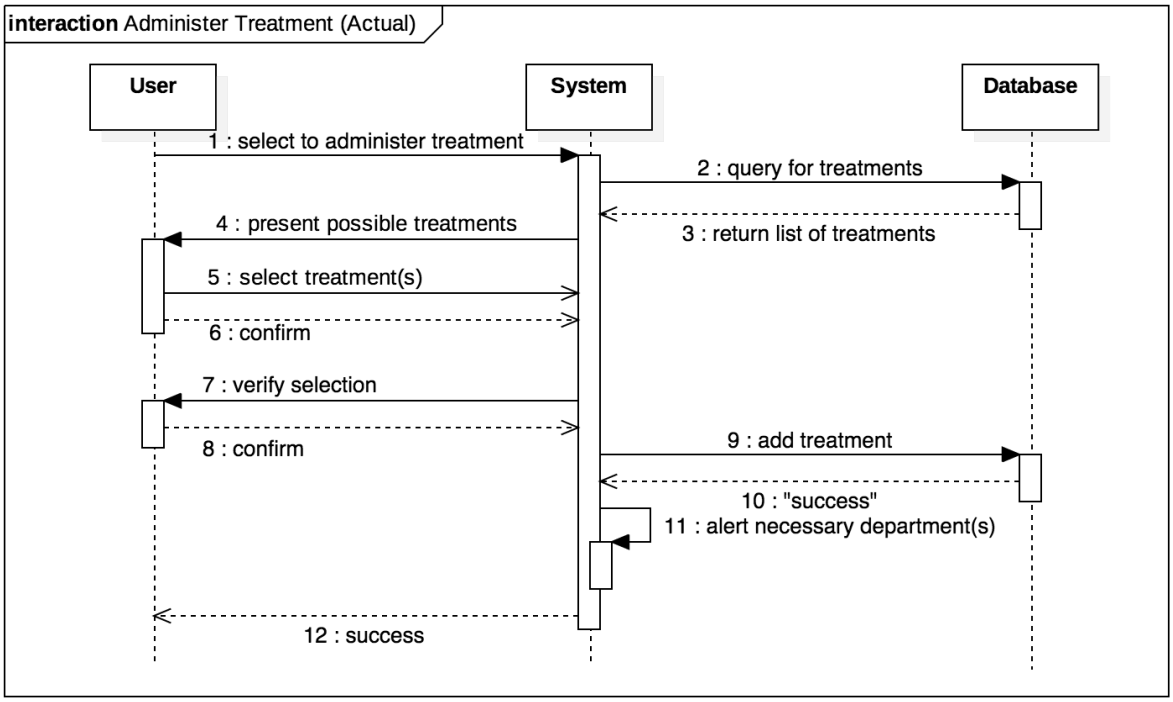

interaction Prescribe Medication (Exception ii)
User
System
Database
1 : select to prescribe medication
2 : query for medications
3 : return list of medicines
4 : present possible medications
5 : select medicine(s)
6 : select dosage(s)
7 : confirm
8 : verify selection
9 : confirm
10 : add prescription
11 : "failure"
12 : failed to add prescription

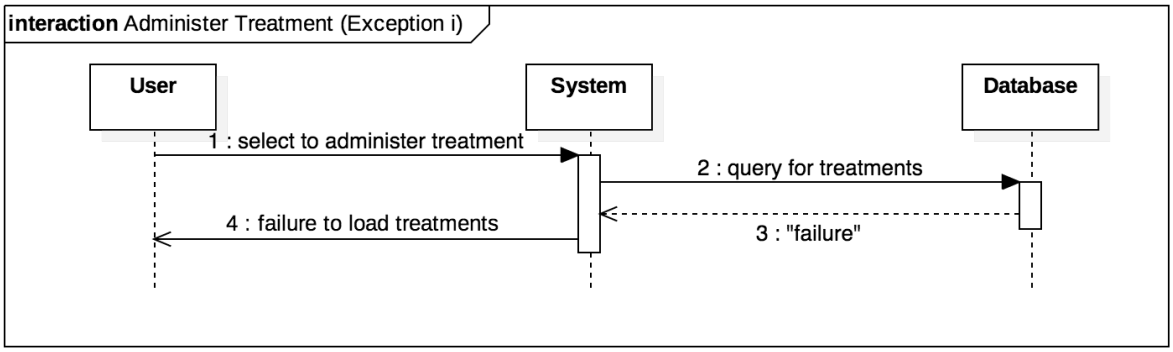

interaction Request Prescription
«Actor» User
«boundary» Dashboard view
«boundary» Display Prescription View
«controller» HTTP Controller
«entity» Patient
«Controller» Database manager
1 : choose select prescriptions
2 : display
3 : enter search criteria
4 : send criteria
5 : query
6 : return matching patients
7 : return results
8 : display
9 : mark prescriptions and quantity
10 : save scripts
«create»
11 : new prescription
«entity» Prescription
12 : confirm scripts
13 : confirm
14 : save info
15 : save info
16 : update
«create»
17 : notification
«entity» notification
18 : query parties
19 : return results
20 : send notifications
21 : update with success

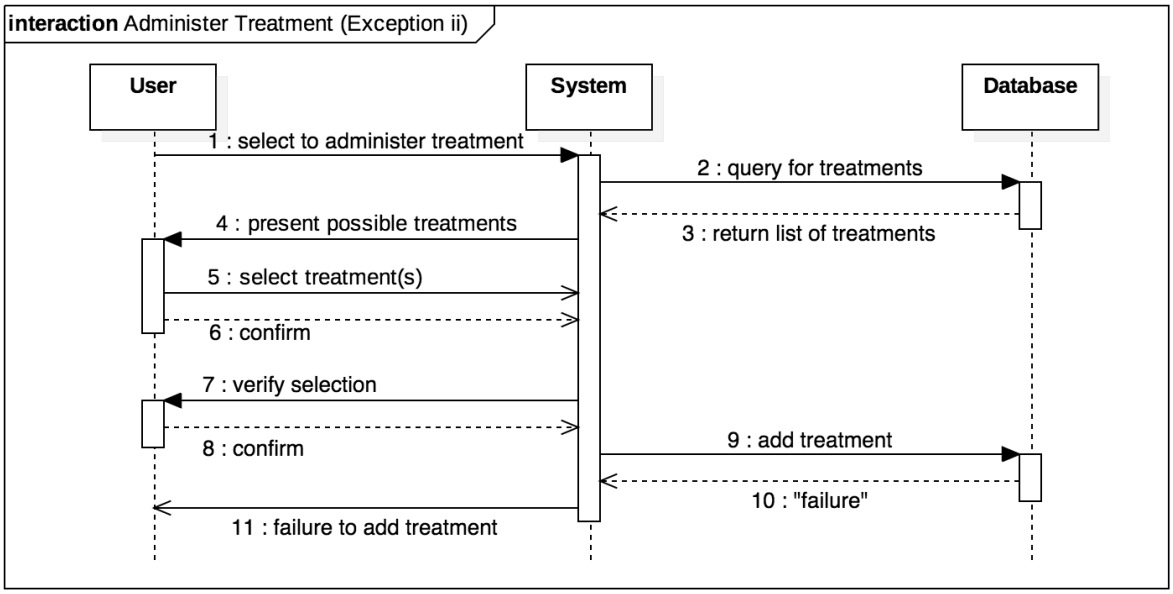
11. **Administer Treatment**
    a. **Participants:** Operational User
    b. **Preconditions**
        i. The user is logged in
        ii. The user is viewing a patient record
    c. **Typical Course of Events**

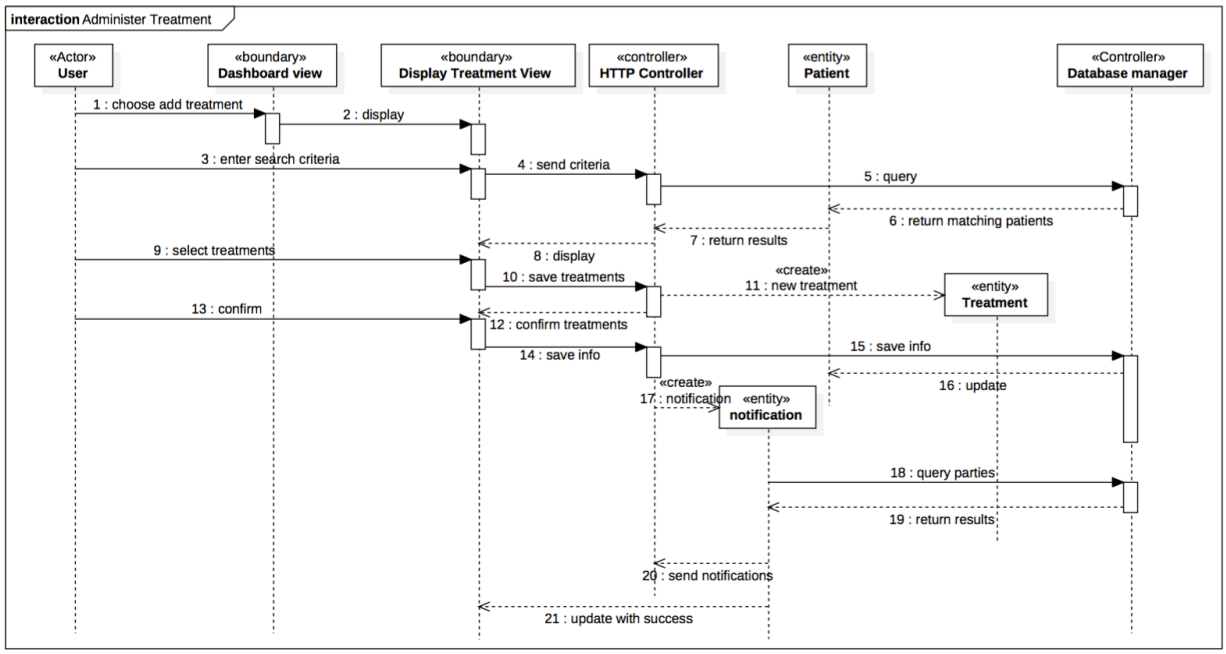
Actor Intentions	System Responsibilities
1. User indicates they want to administer a treatment	
	2. The system presents the user with a list of treatments that can be given
3. The user selects the treatment(s) they want to give to their patient	
	4. The system asks the user to double check the selection
5. The user confirms	
	6. The system saves the new treatment and alerts any necessary departments

    d. **Alternative Courses**
        i. **Step 4:** The system is unable to load the list of treatments
        ii. **Step 6:** The system is unable to save the information, and lets the user know
    e. **Postcondition:** The necessary treatments have been added to the patient's record and are viewable to anyone else with access


interaction Administer Treatment (Actual)
User
System
Database
1 : select to administer treatment
2 : query for treatments
3 : return list of treatments
4 : present possible treatments
5 : select treatment(s)
6 : confirm
7 : verify selection
8 : confirm
9 : add treatment
10 : "success"
11 : alert necessary department(s)
12 : success


interaction Administer Treatment (Exception i)
User
System
Database
1 : select to administer treatment
2 : query for treatments
3 : "failure"
4 : failure to load treatments

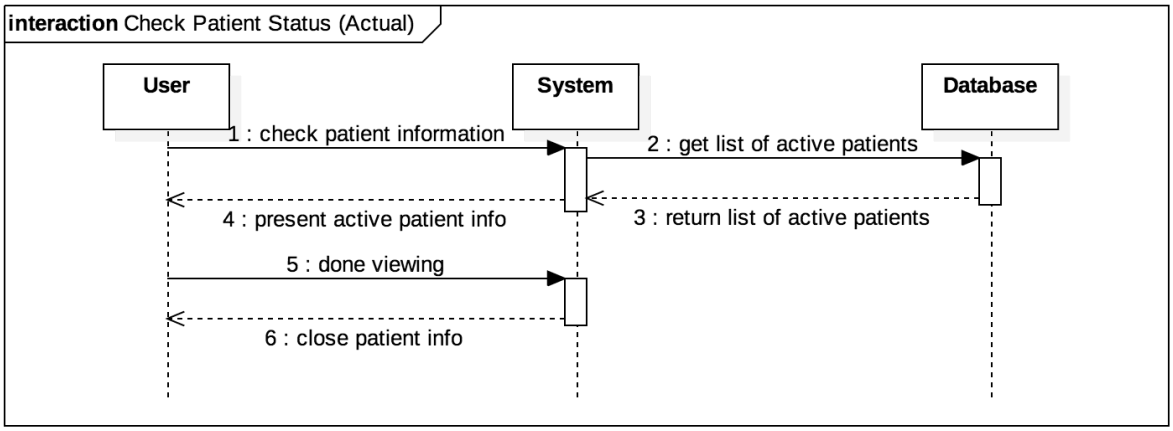

interaction Administer Treatment (Exception ii)
User
System
Database
1 : select to administer treatment
2 : query for treatments
3 : return list of treatments
4 : present possible treatments
5 : select treatment(s)
6 : confirm
7 : verify selection
8 : confirm
9 : add treatment
10 : "failure"
11 : failure to add treatment

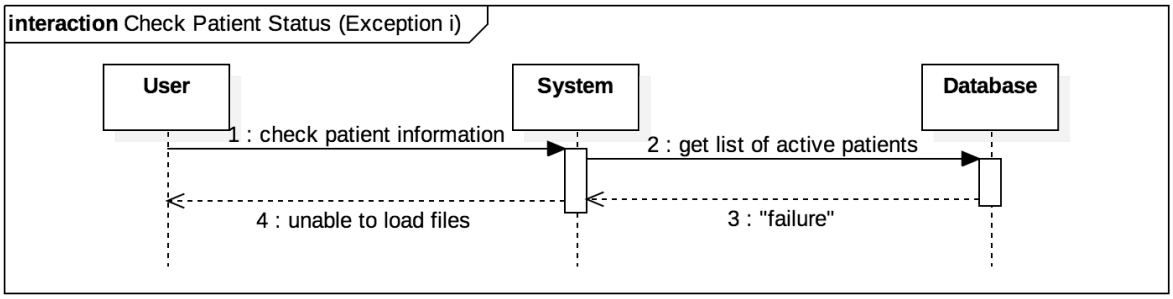

interaction Administer Treatment
«Actor»
User
«boundary»
Dashboard view
«boundary»
Display Treatment View
«controller»
HTTP Controller
«entity»
Patient
«Controller»
Database manager
1 : choose add treatment
2 : display
3 : enter search criteria
4 : send criteria
5 : query
6 : return matching patients
7 : return results
8 : display
9 : select treatments
10 : save treatments
«create»
11 : new treatment
«entity»
Treatment
12 : confirm treatments
13 : confirm
14 : save info
15 : save info
16 : update
«create»
17 : notification
«entity»
notification
18 : query parties
19 : return results
20 : send notifications
21 : update with success

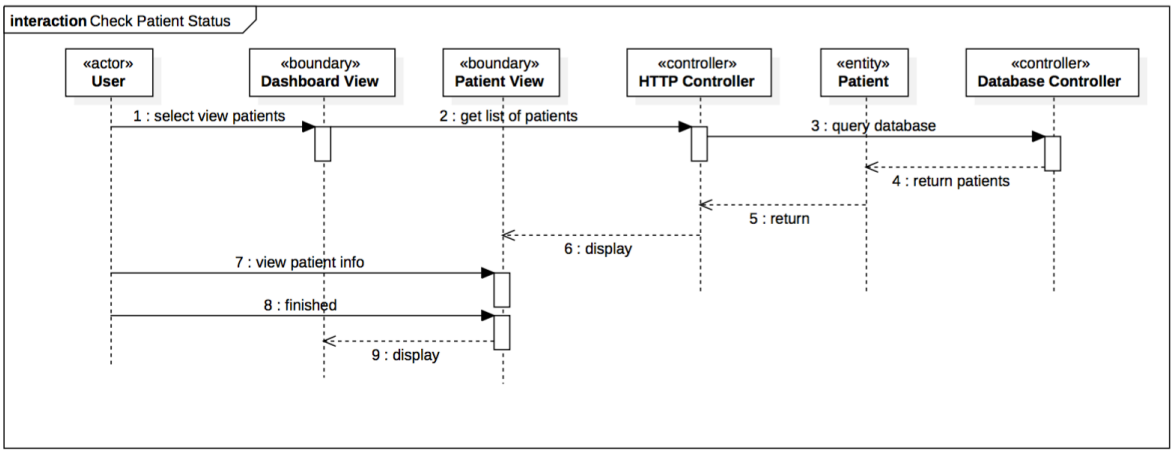
12. **Check Patient Status**
    a. **Participants:** Technical user, Operational user
    b. **Preconditions**
        i. The user is logged in
        ii. The user has a patient in their care
    c. **Typical Course of Events**

Actor Intentions	System Responsibilities
1. User presses the button to check the status of their patients	
	2. The system finds all of the patients whose doctor matches the user and presents a list of the patients
3. The user browses the list of patients and can see vital signs and recent information	
4. The user indicates they are finished viewing the status of their patients	
	5. The system closes the list of patients

    d. **Alternative Courses**
        i. **Step 2:** The database fails to load information
    e. **Postcondition:** The user has seen the most recent information of the patients they are in care of


interaction Check Patient Status (Actual)
User
System
Database
1 : check patient information
2 : get list of active patients
4 : present active patient info
3 : return list of active patients
5 : done viewing
6 : close patient info

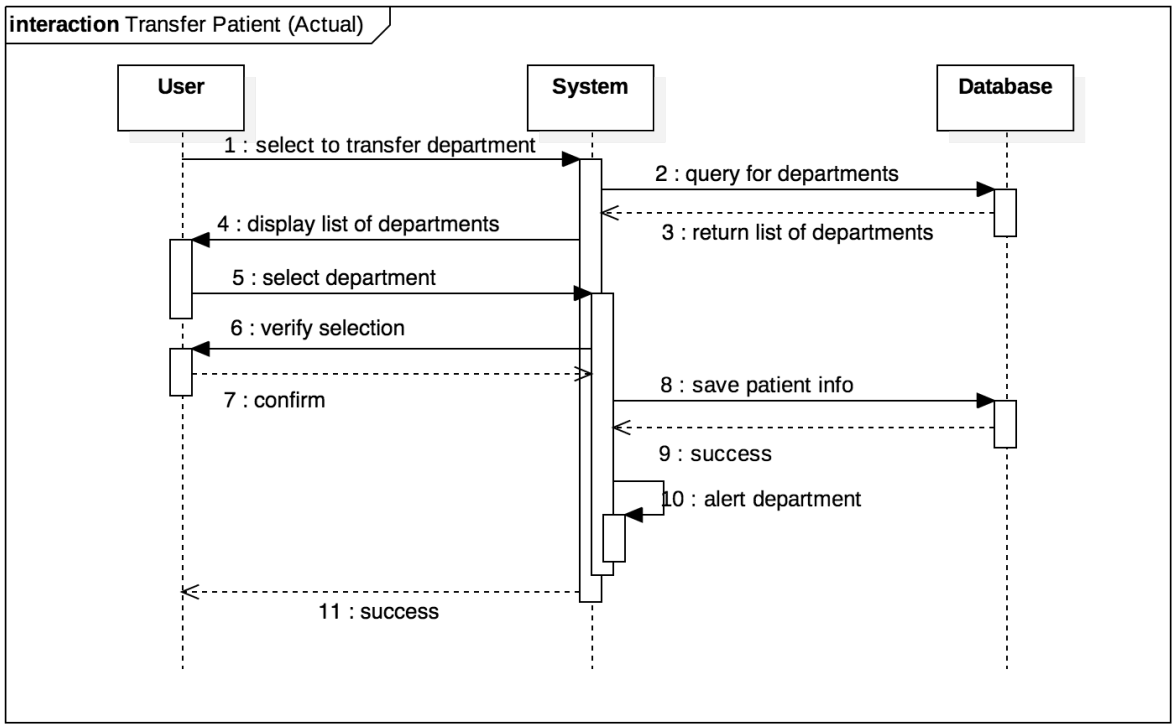

interaction Check Patient Status (Exception i)
User
System
Database
1 : check patient information
2 : get list of active patients
4 : unable to load files
3 : "failure"

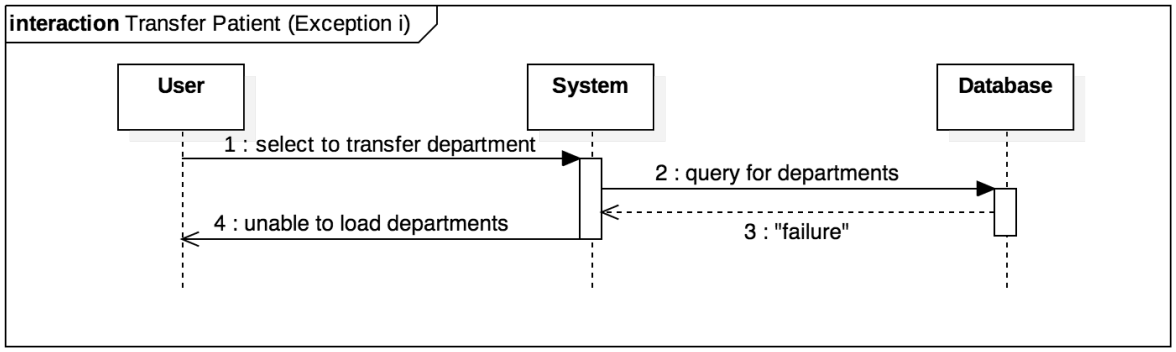

interaction Check Patient Status
«actor» User
«boundary» Dashboard View
«boundary» Patient View
«controller» HTTP Controller
«entity» Patient
«controller» Database Controller
1 : select view patients
2 : get list of patients
3 : query database
4 : return patients
5 : return
6 : display
7 : view patient info
8 : finished
9 : display

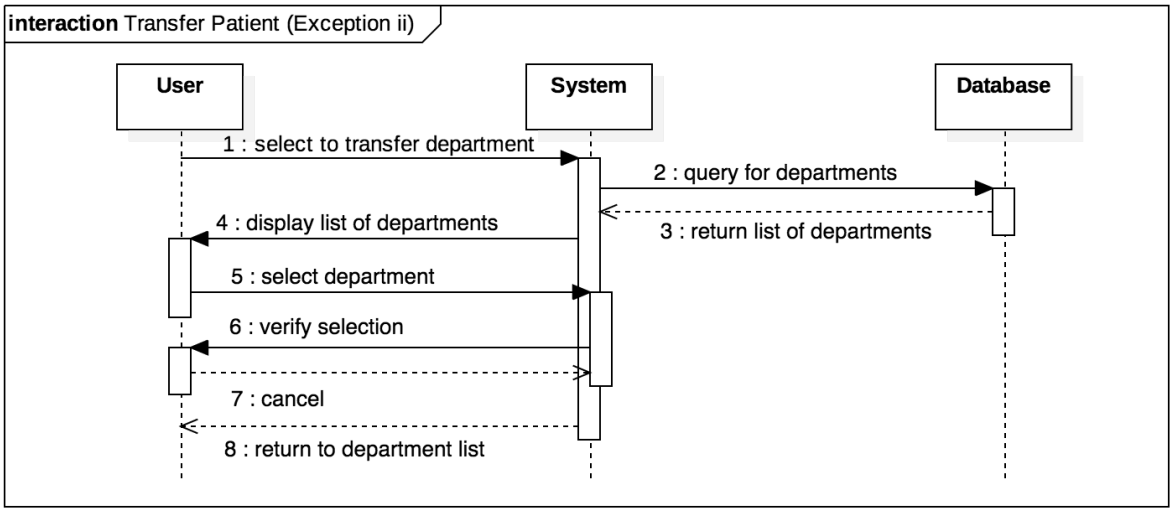
13. **Transfer Patient**
    a. **Participants:** Operational user
    b. **Preconditions**
        i. The user is logged in
        ii. The user is viewing a patient file
        iii. There is another department to be transferred to
    c. **Typical Course of Events**

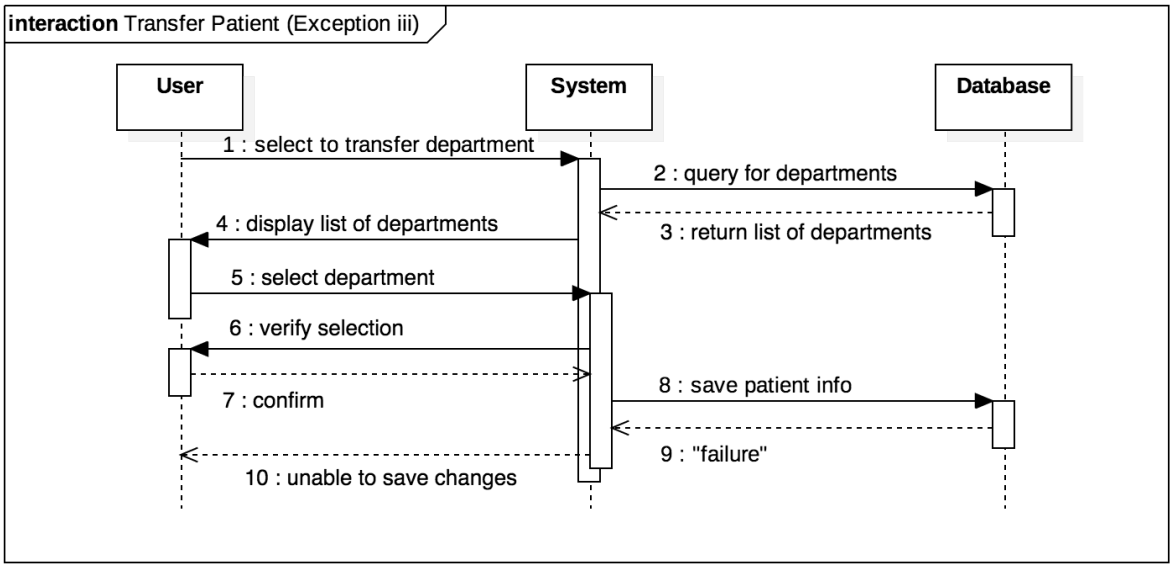
Actor Intentions	System Responsibilities
1. User indicates they would like to refer a file to another department	
	2. The system presents the user with a list of departments the patient can be transferred to
3. The user selects the department they want to transfer the patient to	
	4. The system asks the user to double check the selection
5. The system saves the updated patient record and assigns a new doctor to the patient	
	6. The system saves the updated record and assigns a new doctor to the patient

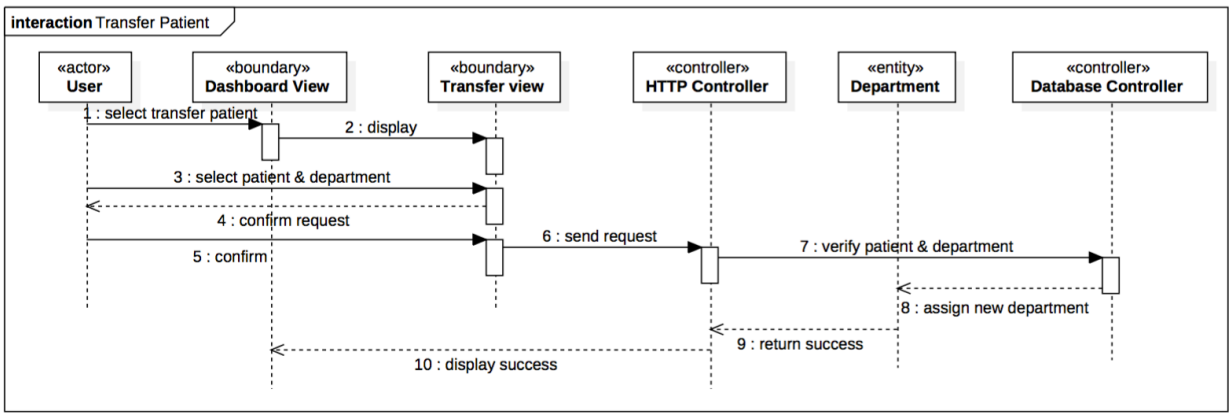
    d. **Alternative Courses**
        i. **Step 2:** The system is unable to load the list of departments
        ii. **Step 4:** The user realizes they made a mistake and need to make changes to the newly assigned department
        iii. **Step 5:** The system is unable to save the new information or assign a new doctor
    e. **Postcondition:** The patient has been reassigned to a new department based on their previous doctor's request


interaction Transfer Patient (Actual)
User
System
Database
1 : select to transfer department
2 : query for departments
3 : return list of departments
4 : display list of departments
5 : select department
6 : verify selection
7 : confirm
8 : save patient info
9 : success
10 : alert department
11 : success


interaction Transfer Patient (Exception i)
User
System
Database
1 : select to transfer department
2 : query for departments
3 : "failure"
4 : unable to load departments


interaction Transfer Patient (Exception ii)
User
System
Database
1 : select to transfer department
2 : query for departments
3 : return list of departments
4 : display list of departments
5 : select department
6 : verify selection
7 : cancel
8 : return to department list

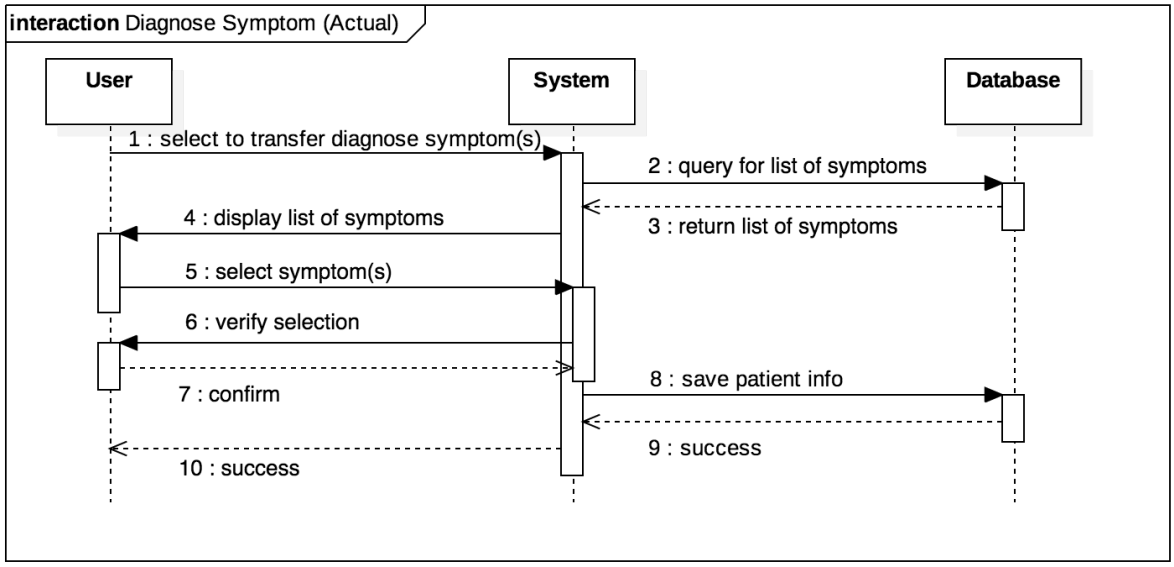

interaction Transfer Patient (Exception iii)
User
System
Database
1 : select to transfer department
2 : query for departments
3 : return list of departments
4 : display list of departments
5 : select department
6 : verify selection
7 : confirm
8 : save patient info
9 : "failure"
10 : unable to save changes

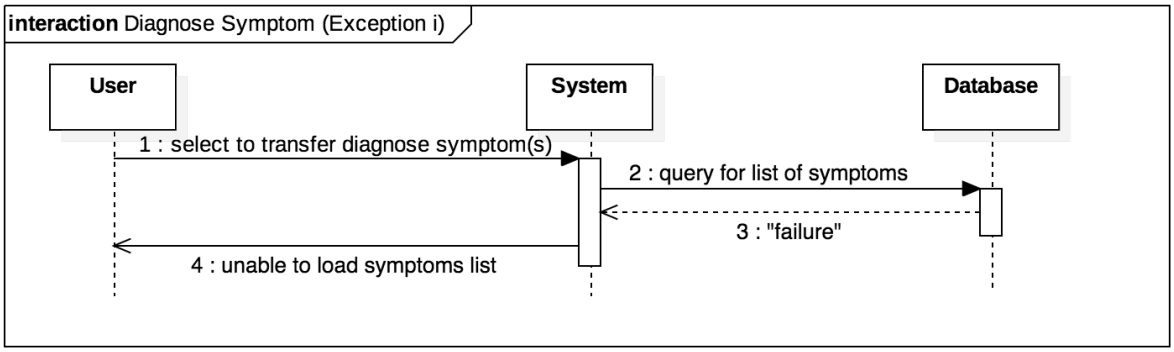

interaction Transfer Patient
«actor» User
«boundary» Dashboard View
«boundary» Transfer view
«controller» HTTP Controller
«entity» Department
«controller» Database Controller
1 : select transfer patient
2 : display
3 : select patient & department
4 : confirm request
5 : confirm
6 : send request
7 : verify patient & department
8 : assign new department
9 : return success
10 : display success

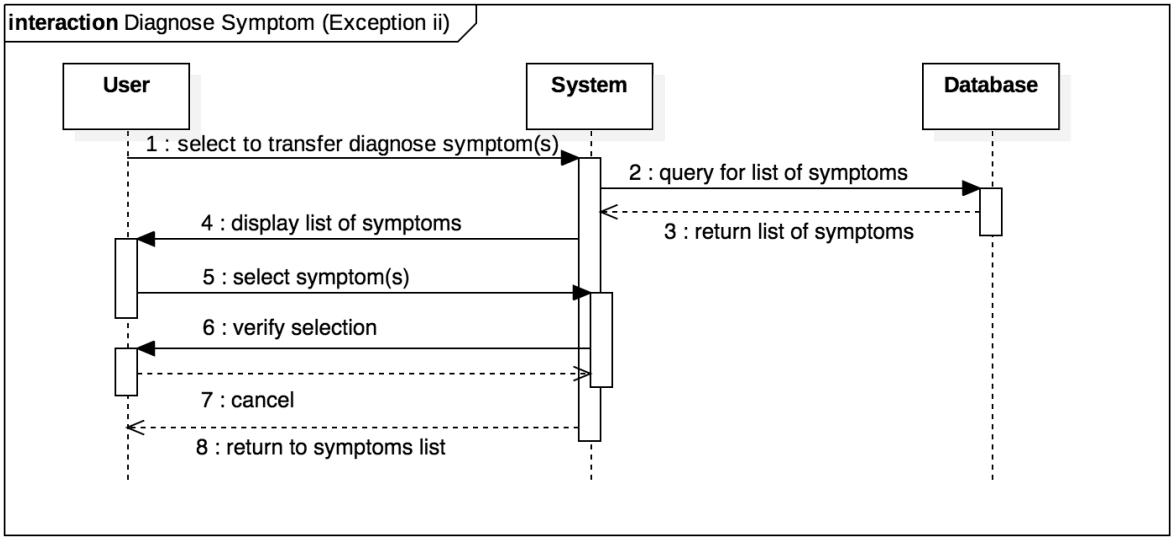
14. **Diagnose Symptom**
    a. **Participants:** Technical user, Operational user
    b. **Preconditions**
        i. The user is logged in
        ii. The user is viewing a patient record
    c. **Typical Course of Events**

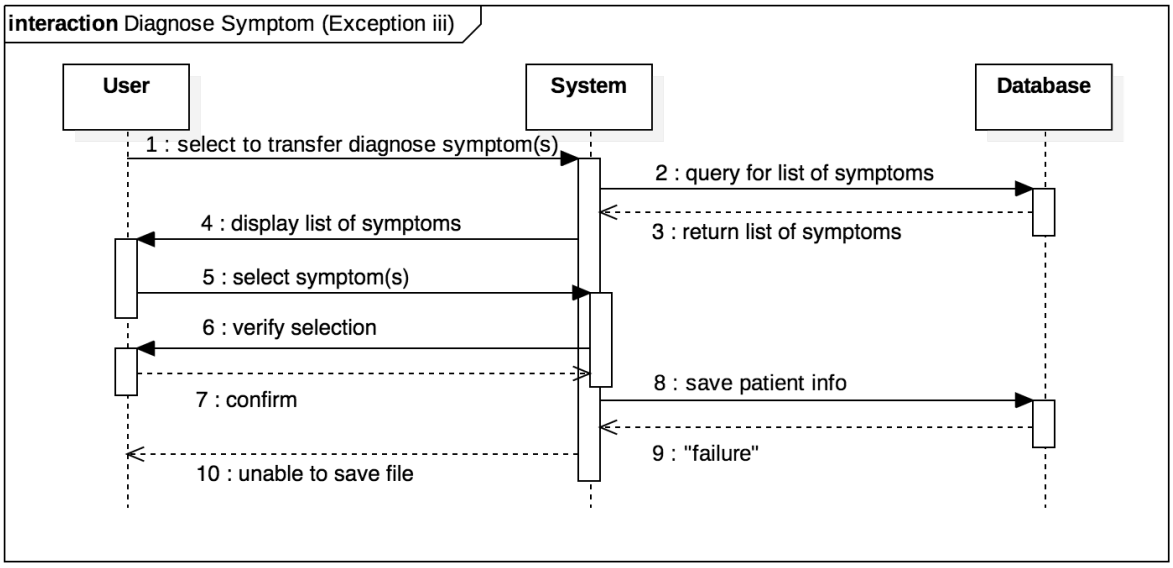
Actor Intentions	System Responsibilities
1. User selects to diagnose a new symptom(s)	
	2. The system presents the user with a list of symptoms to add to the patient's record
3. The user selects the symptoms their patient has indicated are new	
	4. The system asks the user to double check the selection
	5. The system saves the new symptom(s) to the patient's record

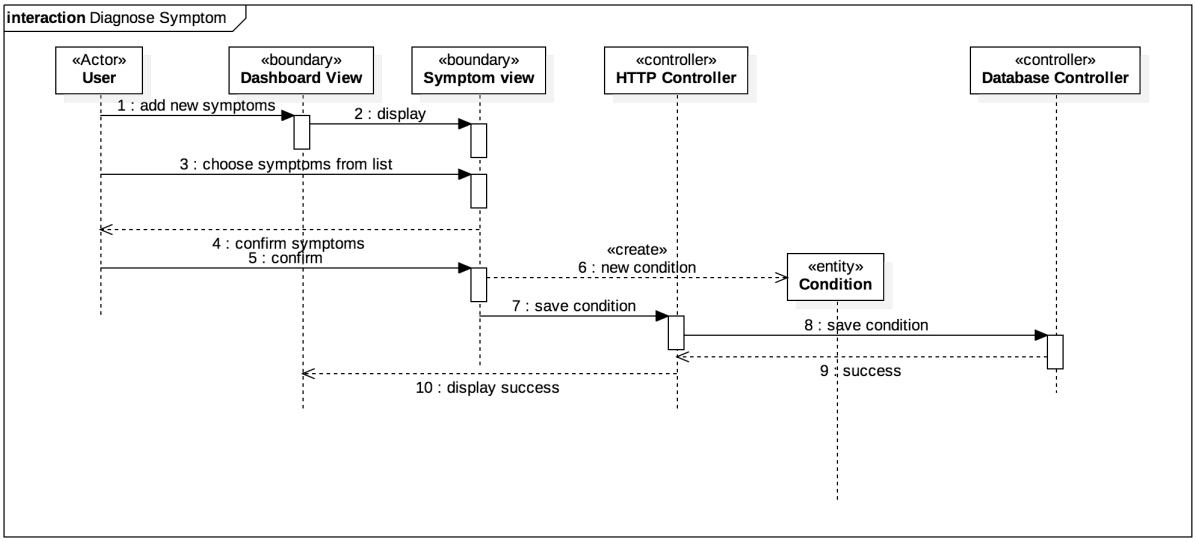
    d. **Alternative Courses**
        i. **Step 2:** The system is unable to load the list of symptoms
        ii. **Step 4:** The user realizes they made a mistake and need to make changes to the new symptoms
        iii. **Step 5:** The system is unable to save the new symptoms to the patient's record


interaction Diagnose Symptom (Actual)
User
System
Database
1 : select to transfer diagnose symptom(s)
2 : query for list of symptoms
3 : return list of symptoms
4 : display list of symptoms
5 : select symptom(s)
6 : verify selection
7 : confirm
8 : save patient info
9 : success
10 : success


interaction Diagnose Symptom (Exception i)
User
System
Database
1 : select to transfer diagnose symptom(s)
2 : query for list of symptoms
3 : "failure"
4 : unable to load symptoms list


interaction Diagnose Symptom (Exception ii)
User
System
Database
1 : select to transfer diagnose symptom(s)
2 : query for list of symptoms
3 : return list of symptoms
4 : display list of symptoms
5 : select symptom(s)
6 : verify selection
7 : cancel
8 : return to symptoms list

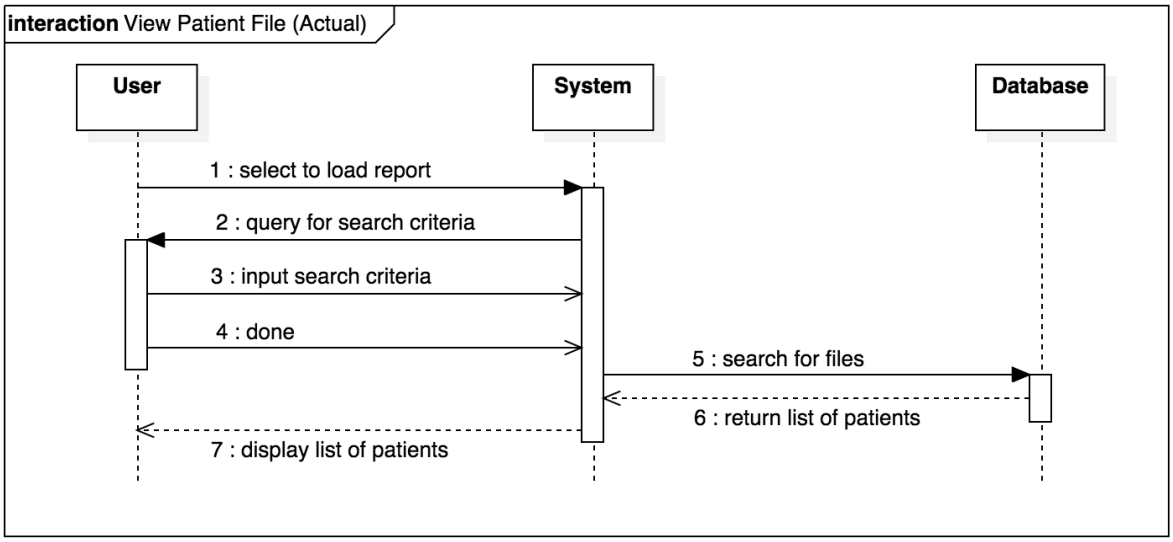

interaction Diagnose Symptom (Exception iii)
User
System
Database
1 : select to transfer diagnose symptom(s)
2 : query for list of symptoms
3 : return list of symptoms
4 : display list of symptoms
5 : select symptom(s)
6 : verify selection
7 : confirm
8 : save patient info
9 : "failure"
10 : unable to save file

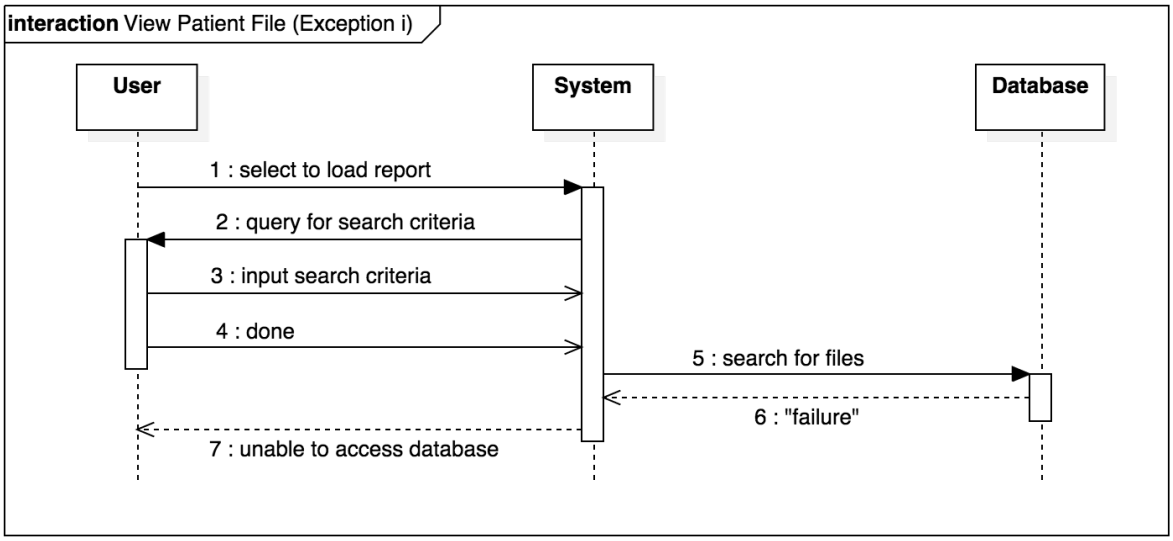

interaction Diagnose Symptom
«Actor» User
«boundary» Dashboard View
«boundary» Symptom view
«controller» HTTP Controller
«controller» Database Controller
«entity» Condition
1 : add new symptoms
2 : display
3 : choose symptoms from list
4 : confirm symptoms
5 : confirm
«create»
6 : new condition
7 : save condition
8 : save condition
9 : success
10 : display success

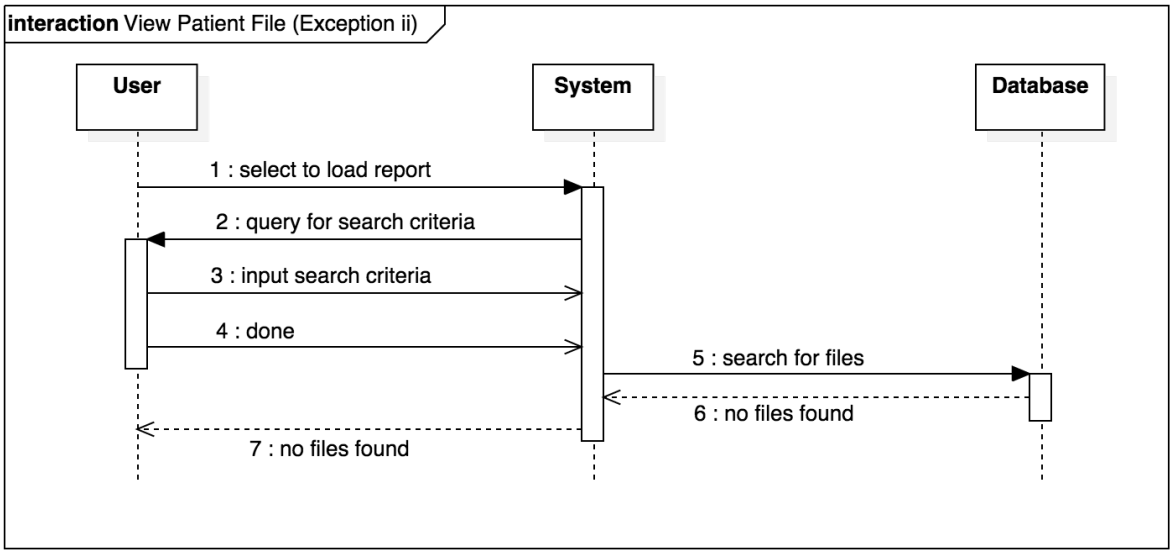
15. **View Patient Record**
    a. **Participants:** Administrative user, Technical user, Operational user, Database
    b. **Preconditions**
        i. The user is logged in
        ii. The user has access to the specific patient
    c. **Typical Course of Events**

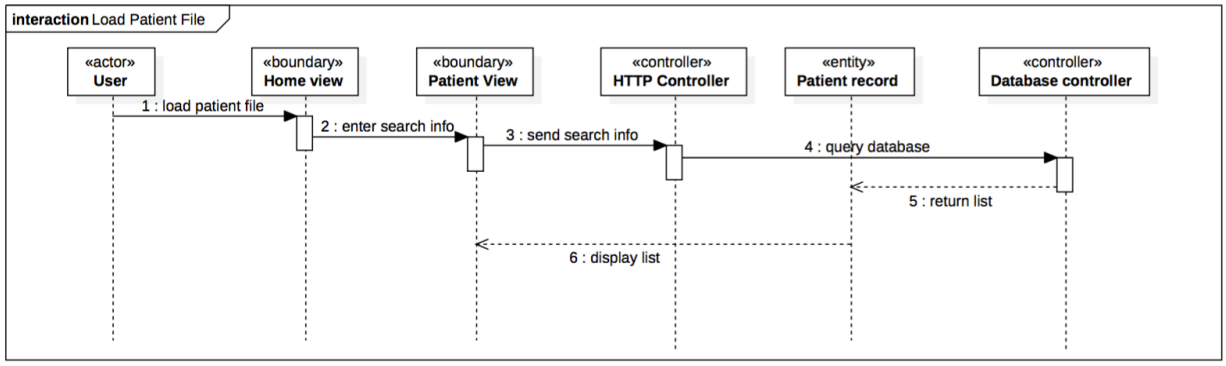
Actor Intentions	System Responsibilities
1. User selects to load a patient record	
	2. The system presents the user with a search box
3. The user inputs search criteria	
	4. The system loads all records that match the criteria
	5. The system displays a list of these records
6. The user selects a patient record	

    d. **Alternative Courses**
        i. **Step 2:** The system is unable to load any records
        ii. **Step 2:** The system is unable to access the database

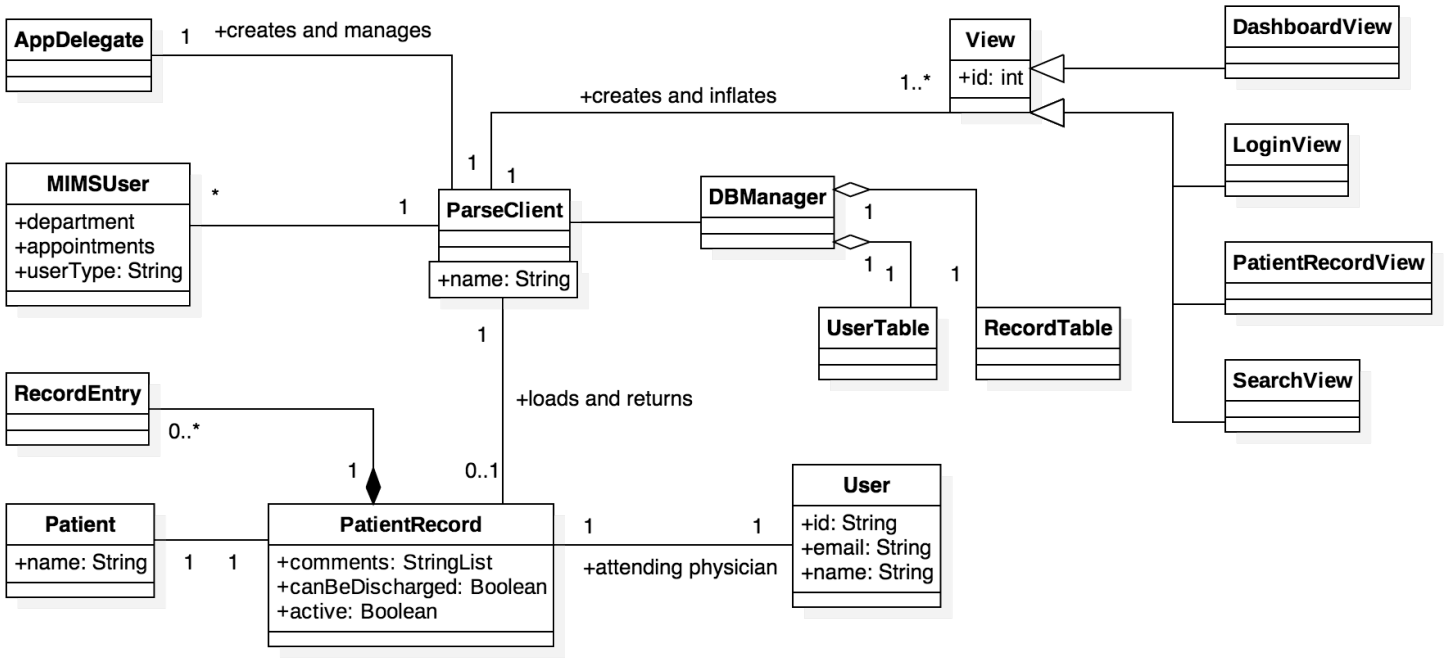

interaction View Patient File (Actual)
User
System
Database
1 : select to load report
2 : query for search criteria
3 : input search criteria
4 : done
5 : search for files
6 : return list of patients
7 : display list of patients


interaction View Patient File (Exception i)
User
System
Database
1 : select to load report
2 : query for search criteria
3 : input search criteria
4 : done
5 : search for files
6 : "failure"
7 : unable to access database

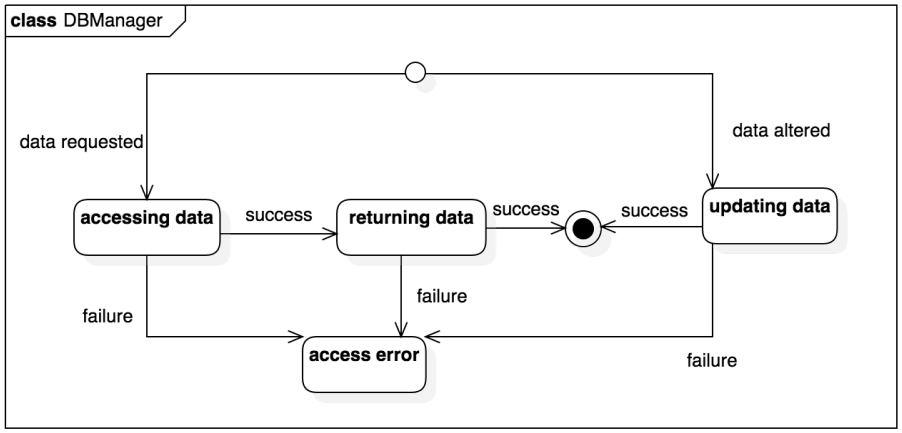

interaction View Patient File (Exception ii)
User
System
Database
1 : select to load report
2 : query for search criteria
3 : input search criteria
4 : done
5 : search for files
6 : no files found
7 : no files found

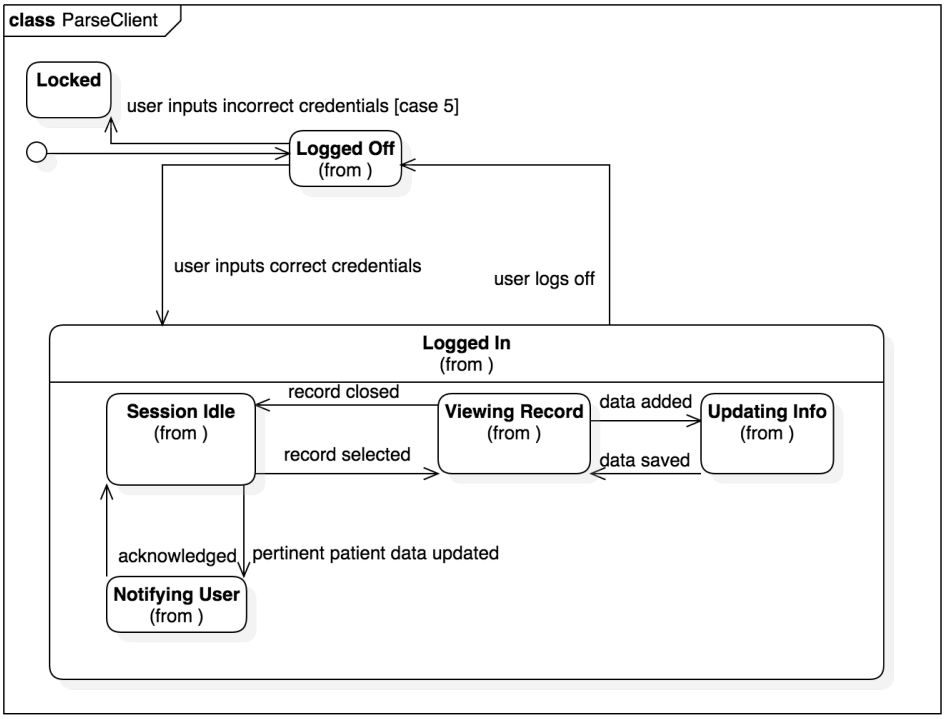

interaction Load Patient File
«actor»
User
«boundary»
Home view
«boundary»
Patient View
«controller»
HTTP Controller
«entity»
Patient record
«controller»
Database controller
1 : load patient file
2 : enter search info
3 : send search info
4 : query database
5 : return list
6 : display list

2.3 Application Class Model

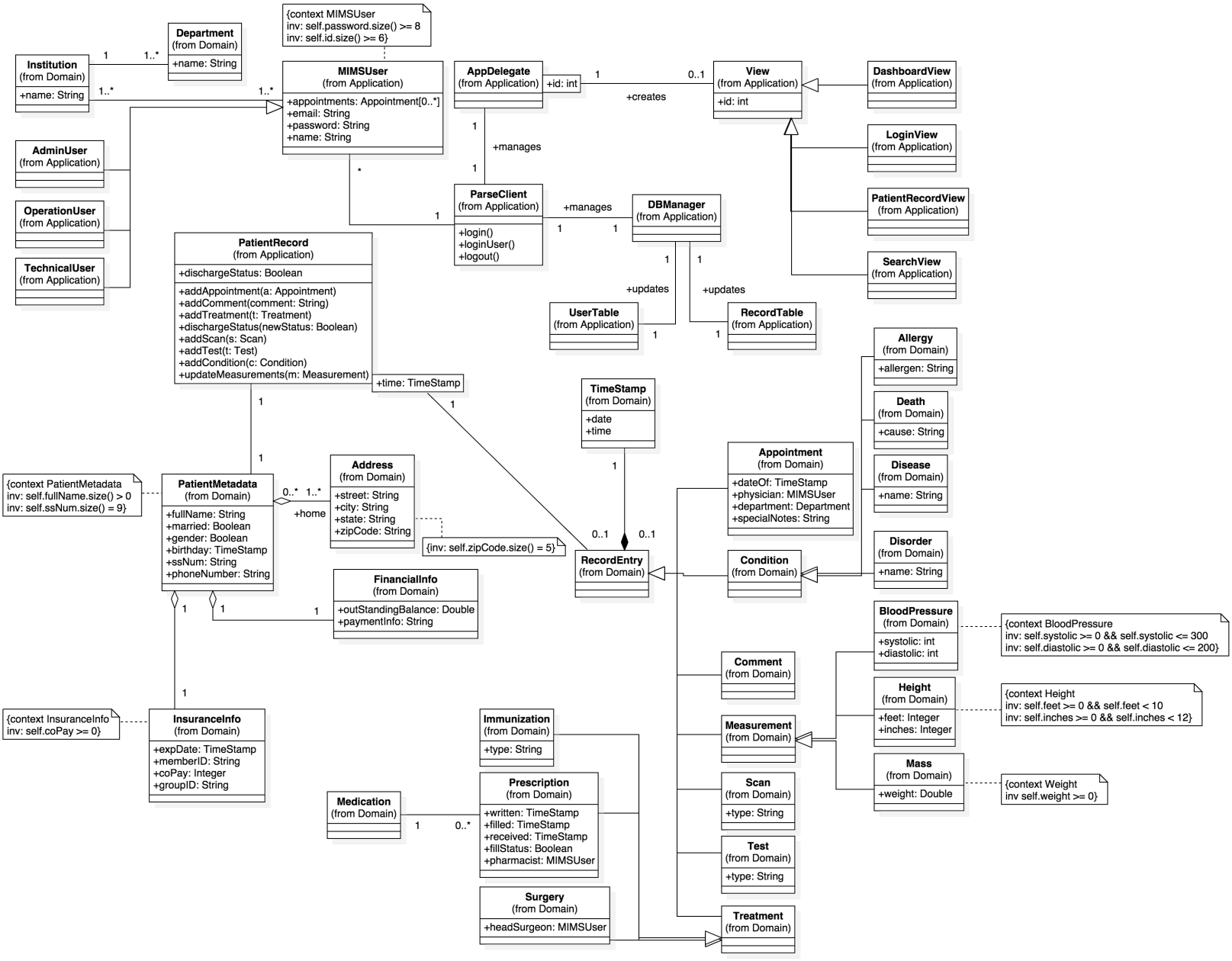


2.4 Application State Model





3 Consolidated Class Model



4 Model Review

Analysis is a crucial stage in the development of any software system. MIMS was analyzed in order to develop a concise, understandable, and correct model for the system. This analysis is broken into two key phases: domain and application.

The domain analysis was concerned with developing a model of real-world components. The focus of domain modeling is on building a model of intrinsic concepts. Physical objects and actors, such as user or computer, were considered when developing the conceptual domain model. Computer specific classes, such as controllers, were not considered during this analysis phase.

The application analysis phase was more rigorous and detailed, and involves adding major application artifacts to the domain model from the previous analysis phase. This phase is heavily concerned with modeling interactions in the system. To start, use cases were derived from the concept statement. Each of these use cases was essential in furthering the analysis. These use cases were later reviewed for correctness. It was found that all use cases had meaningful verb phrases as names and were derived directly from the concept statement. From the use cases, an application interaction model was developed. Normal scenarios were created and activity diagrams were built during this phase. An application class model, which included boundaries and controllers, was developed. Although the domain and application class models have similarities, the application class model considered the application itself, rather than real-world objects, such as users or computers. This application model was developed by specifying interfaces, defining boundary classes, and determining controllers. The class model was then crosschecked against the domain class model. Classes with state were identified and an application state model was developed. At this stage, most classes were found to have meaningful and descriptive names. Furthermore, there was little to no redundancy among classes.

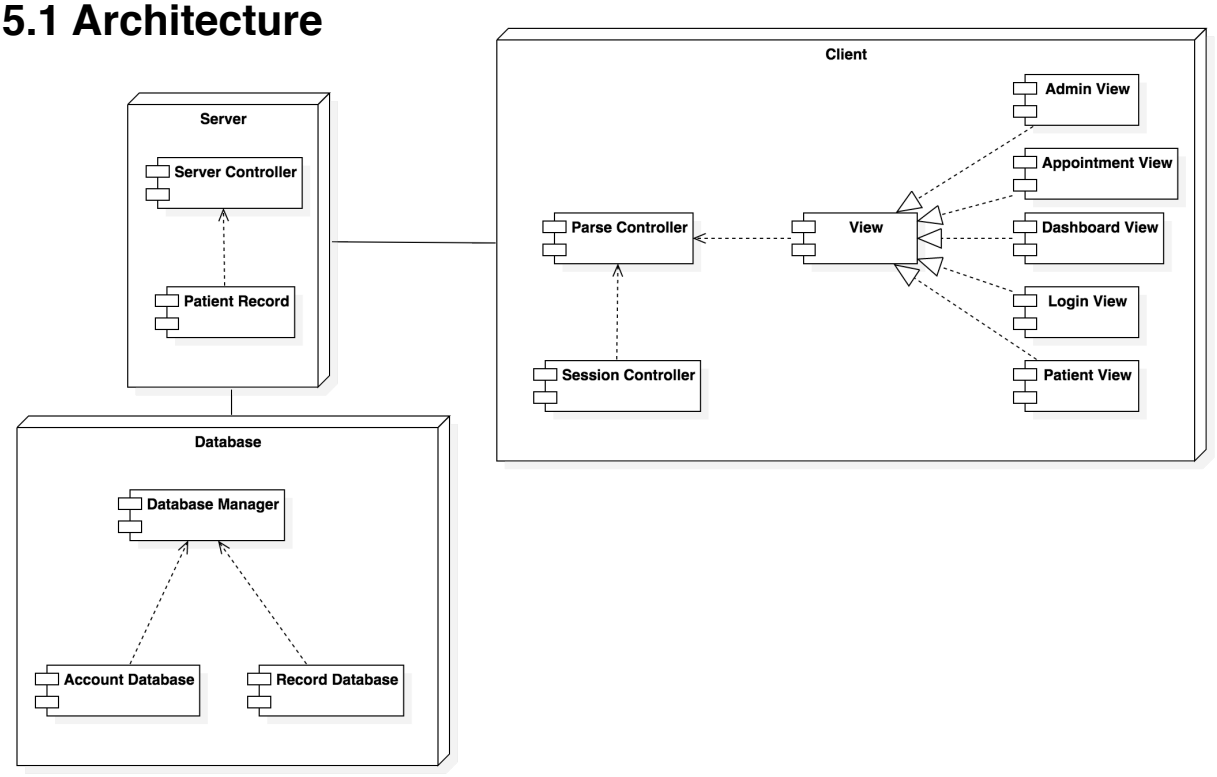
After both the domain and application analysis phases were complete, components from each phase were combined to create a consolidated class model. After this model was completed, inter-consistency between domain and application class models was evaluated. There was some inconsistency between real-world and computer objects, but this was to be expected.

At this stage, classes, associations, attributes, and controllers were evaluated for completeness and correctness. For instance, the domain model included some unnecessary classes that were removed, such as User or Computer. After removing

unnecessary or irrelevant classes, the model was found to contain no duplicate or redundant classes. Controllers were also found to contain necessary associations to access the objects needed. For instance, the SessionController was able to access Session and AccountManager objects. The majority of associations were appropriately specified as one-to-one, with limited aggregation being necessary, with the exception of some of the views. Also, in many instances, the interaction diagrams were less specific than class models. This issue affects the accuracy of the model.

System analysis was complete with the consolidated class model. However, the model is obviously not perfect. Like any model, accuracy improves with continued refinement. Inconsistencies between interaction diagrams and class models may limit the usefulness of the model during the design phase, but the class models are still very accurate and complete.

5.1 Architecture

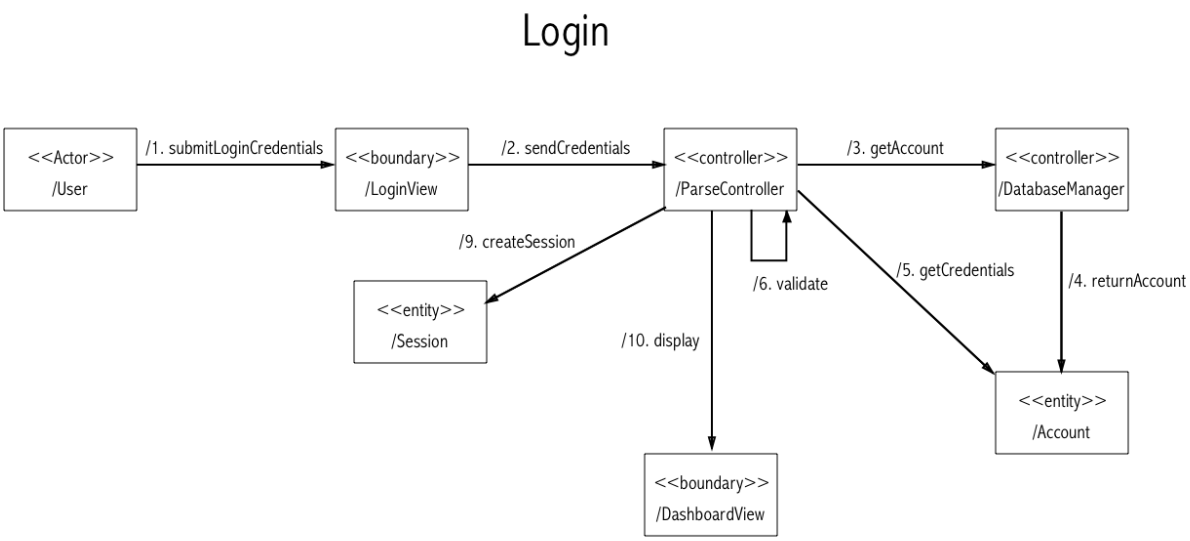


MIMS uses a client server architecture because it's well suited to the style of our app. In our system, a client must retrieve and display object data that is stored on a remote server. Some advantages of using this system include data distribution that is very straightforward and this allows each sub-system to have its own domain among the data. Another advantage involves the user management, which allows us to handle the cases where some users have access that other users don't have based on their roles in MIMS. With this architecture account permissions can be managed from a centralized location (the server), as opposed to having to manage the permissions from a localized location such as the client device. This advantage is even greater considering users can have multiple devices within the hospital (iPhone, iPad, Mac, etc.). Speaking to the advantages of having multiple smart devices, it would also be pertinent for our application to inherit from MVC or Model View Controller architecture. MVC allows each discrete responsibility to be represented separately, where the Model handles all the data store, the View which holds the UI information, and the Controller which mediates the information between the Model and View. This sort of architecture allows us to develop a very functional architecture that’s well suited to mobile applications, especially with multiple views that need to be independent of each other, and separating the models from the view allows us to do this. There are however some disadvantages, such as having to consider the fact that the server could fail or be slow to respond. If the server experiences any downtime the whole system could go down which could be devastating for the user's and the patients. This would not be an issue if all the information was stored on the user's local device (and some of this can be avoided with caching), but obviously this wouldn't be ideal because the user's would need the ability

to have the most up-to-date information at all times. However, because the advantages outweigh the disadvantages, it seems clear that both these patters, MVC and Client-Server, would be the best design pattern for MIMS.

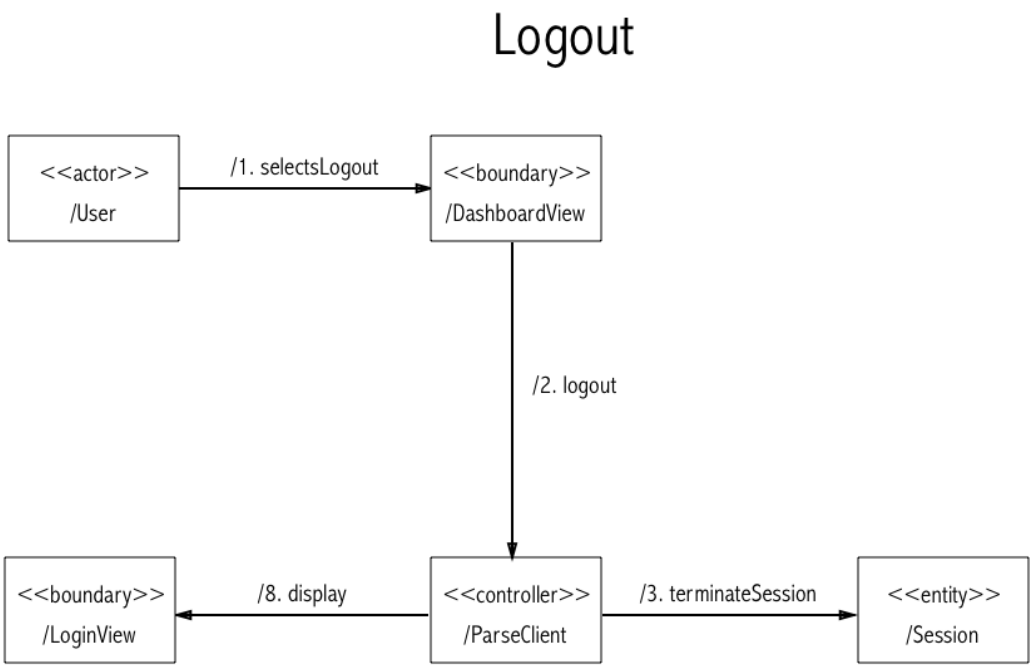
5.2 Interactions

Login



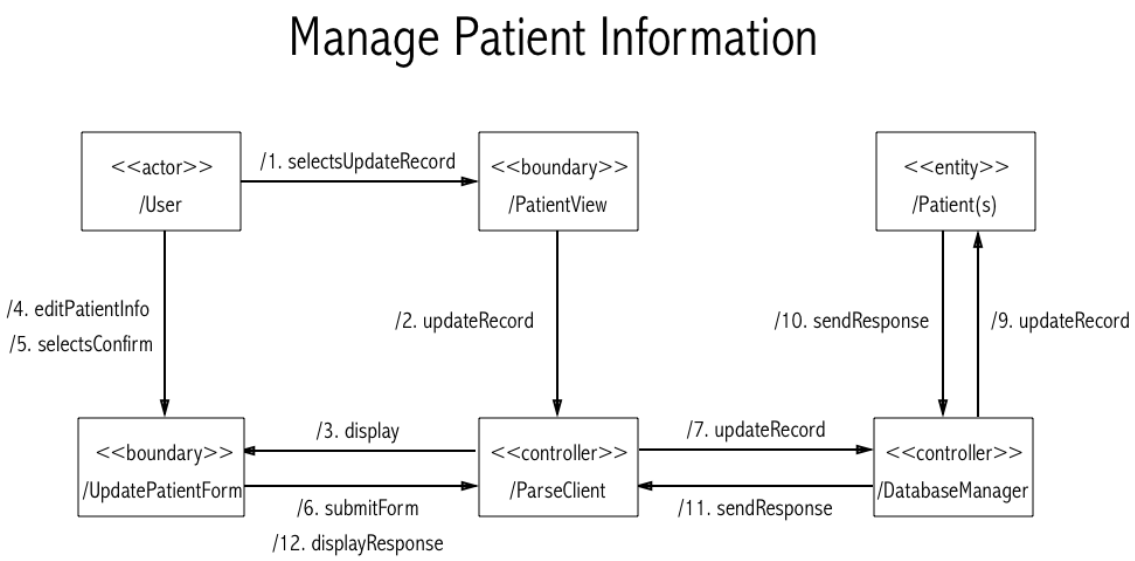
In Login, one of the GRASP patterns used was Controller . This was done in part in the Client subsystem by having the “Client Controller” object handle requests from the UI layer. There is High Cohesion , as each element has highly related responsibilities (in this case, all but Server Controller have just one responsibility each). There is relatively Low Coupling , about as low as could be expected, as each object (aside from ServerController) has a direct connection to just one other object. The Creator pattern is also used, as the ServerController creates the Session entity.

Logout



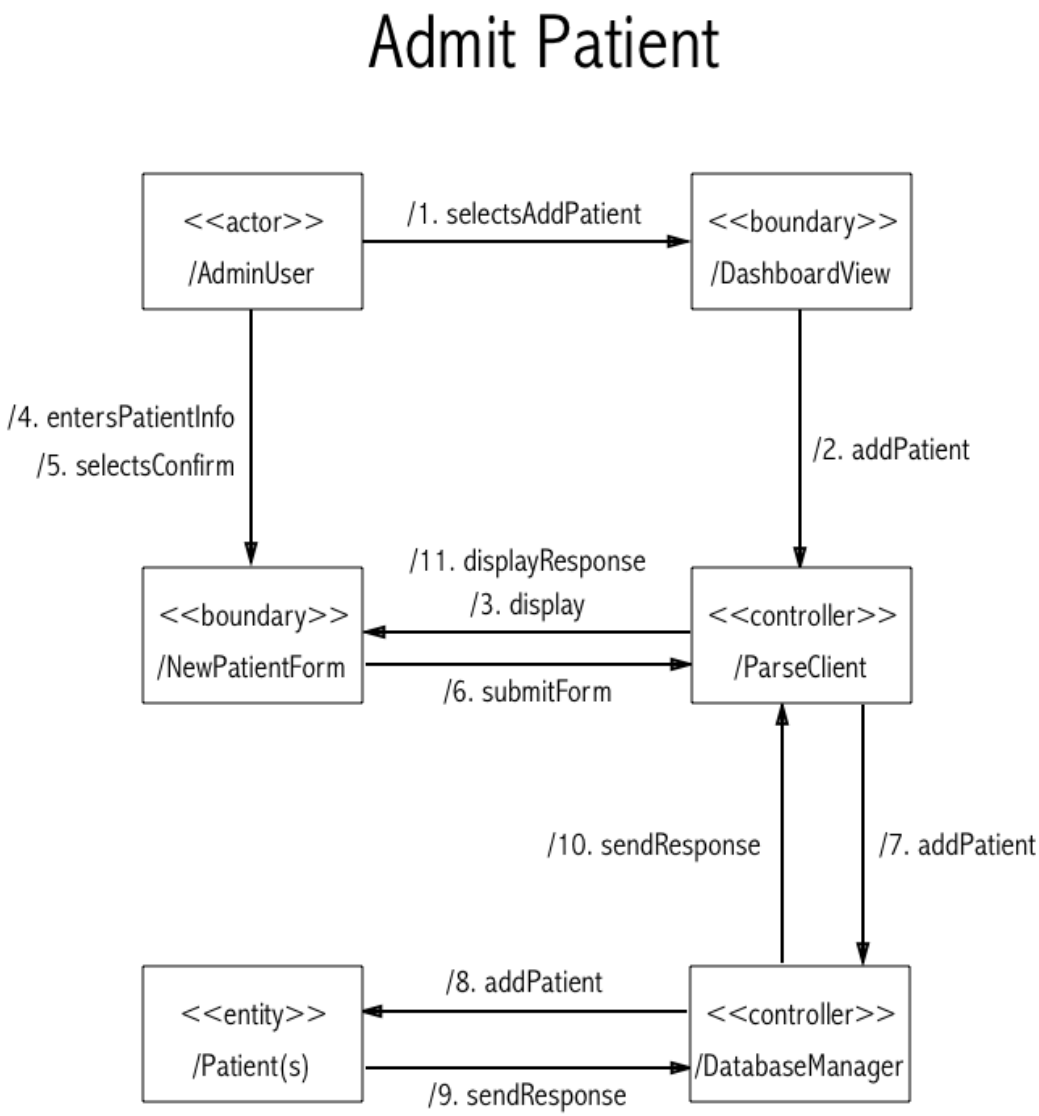
In Logout, one of the GRASP patterns used was Controller. This was done in part in the Client subsystem by having the “Client Controller” object handle requests from the UI layer. There is High Cohesion, as each element has highly related responsibilities (in this case, all but Client Controller have just one responsibility each). There is relatively Low Coupling, about as low as could be expected, as each object (aside from the Client Controller) has a direct connection to just one other object.

Manage Patient Information



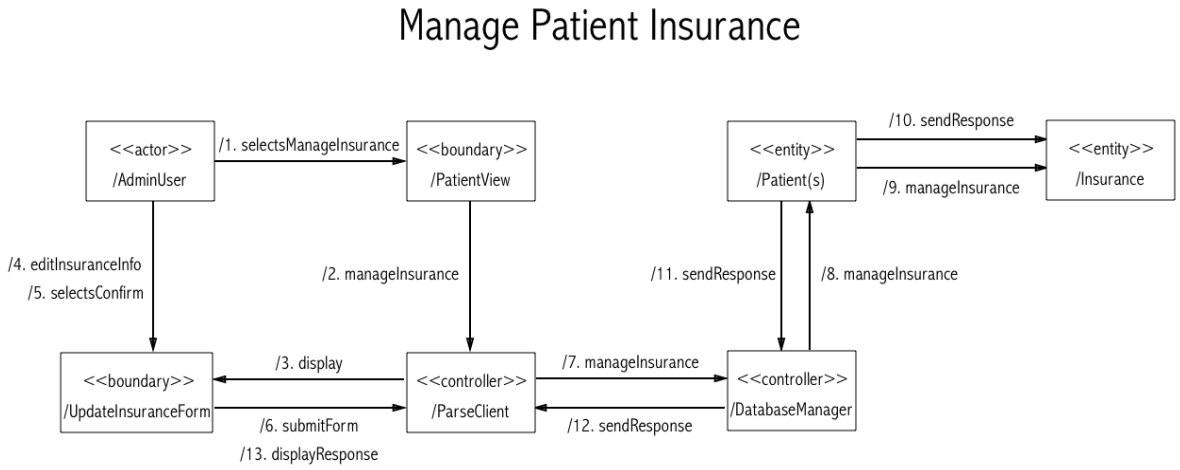
Manage Patient Information mimics the GRASP pattern(s): **Controller, High Cohesion**. The ParseClient controller displays data to the user interface UpdatePatientForm.

Admit Patient



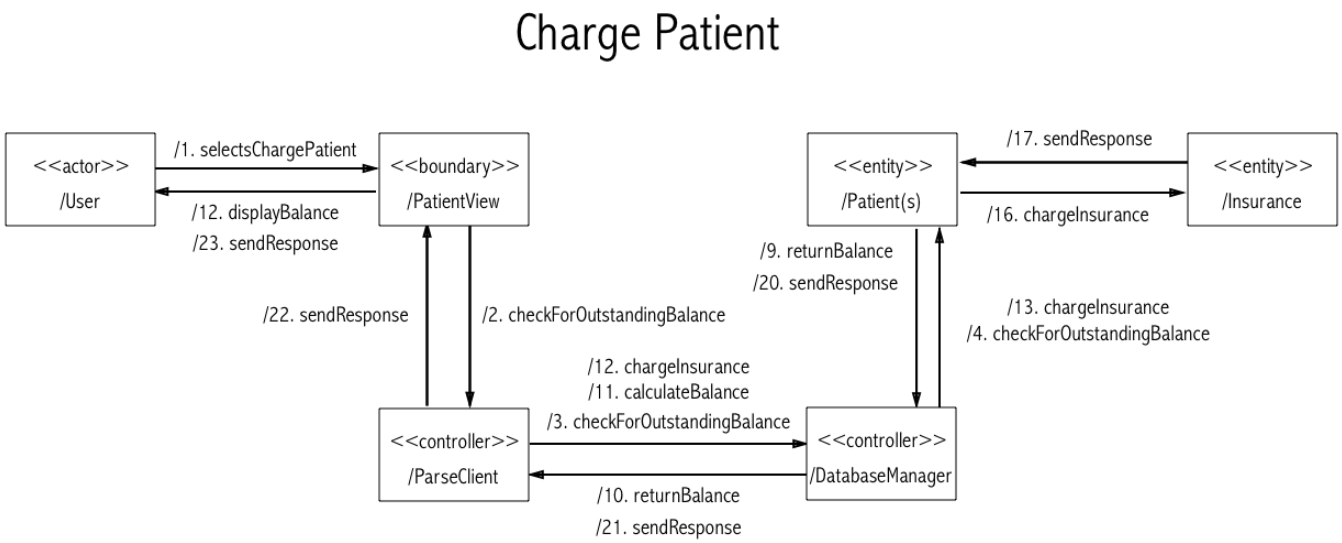
Admit Patient mimics the GRASP pattern: Controller, High Cohesion, Low Coupling, and Pure Fabrication.ParseClient handles information to be displayed in the NewPatientForm user interface and therefore shows controller GRASP patterns. Functions such as, addPatient and sendResponse keeps high cohesion between object interactions. The low coupling is present in this use .The new patient form follows the Pure Fabrication pattern

Manage Patient Insurance Information



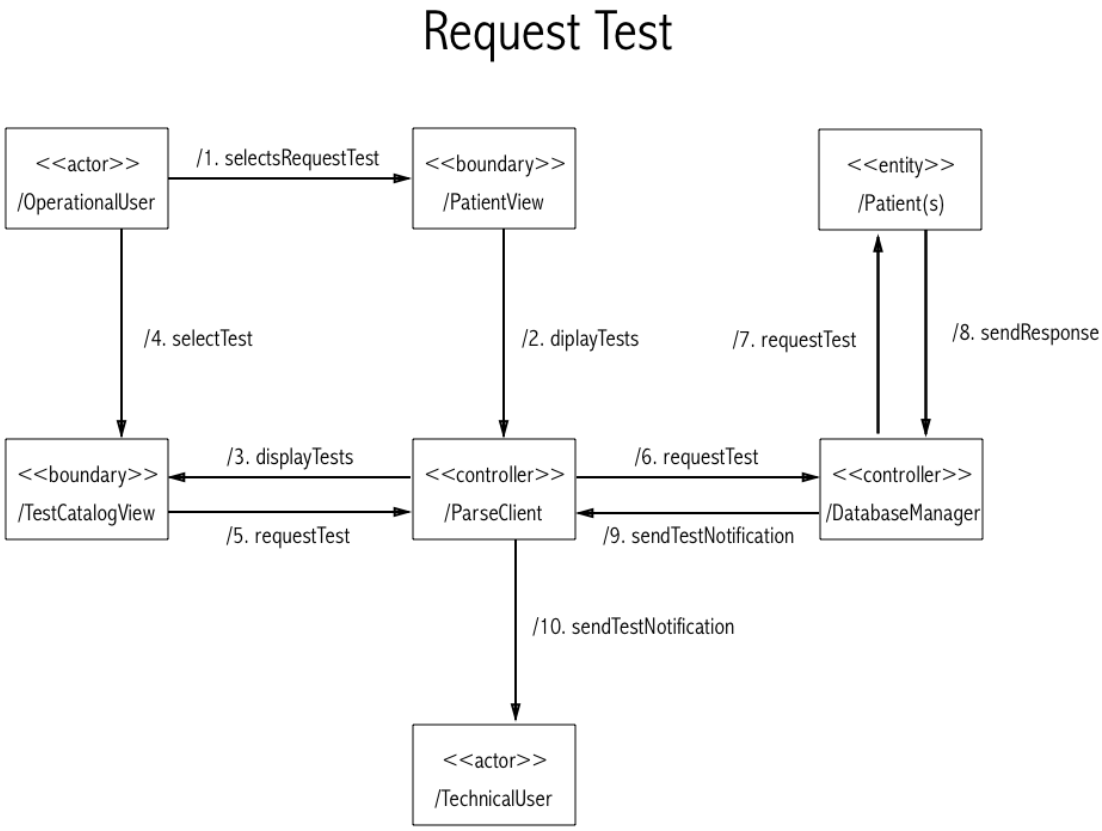
Manage Patient Insurance mimics the GRASP patterns: Controller, High cohesion.
The ParseClient controller displays to the user interface UpdateInsuranceForm. The manageInsurance function keep high cohesion in this use case.

Charge Patient



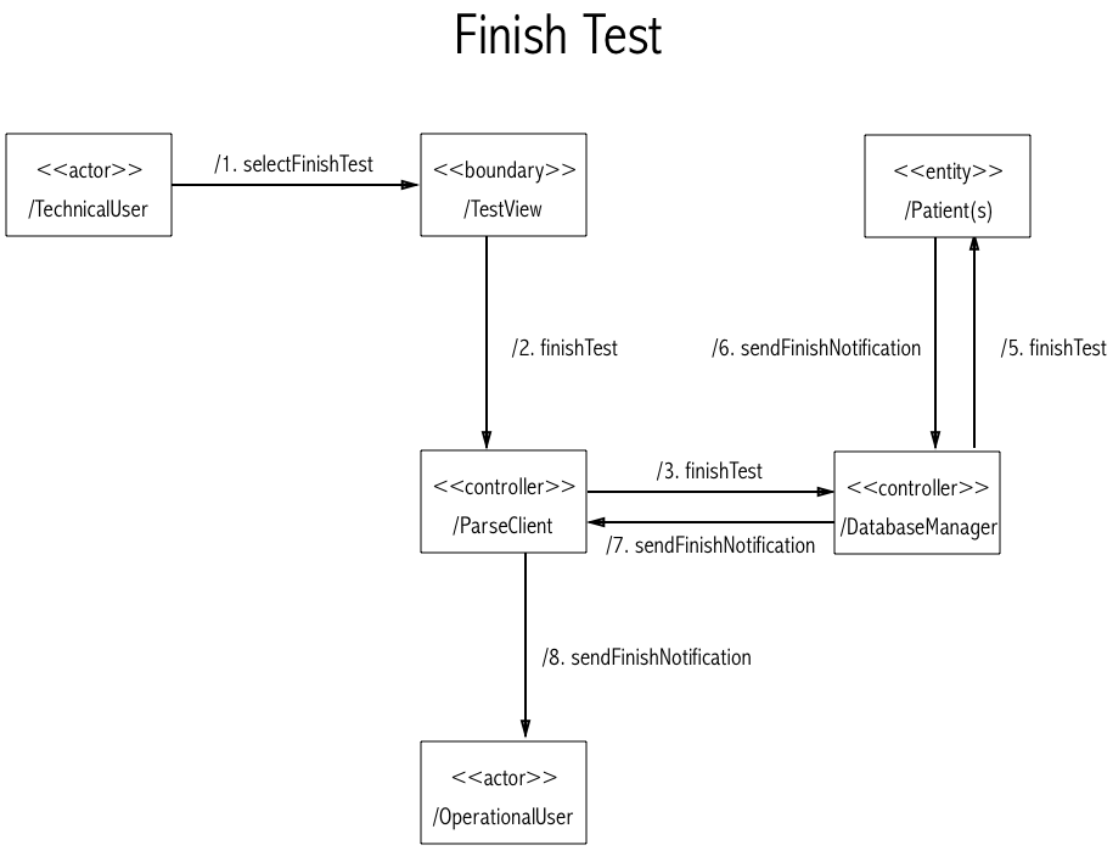
The Charge Patient use case mimic the GRASP patterns: Controller, High Cohesion.
Controller ParseClient handles the display of a response to the PatientView user interface. The DatabaseManager controller handles the UI of Patients entity. Functions such as checkforOutstandingBalance is a common responsibility that is shared between multiple objects in this use case.

Request Test



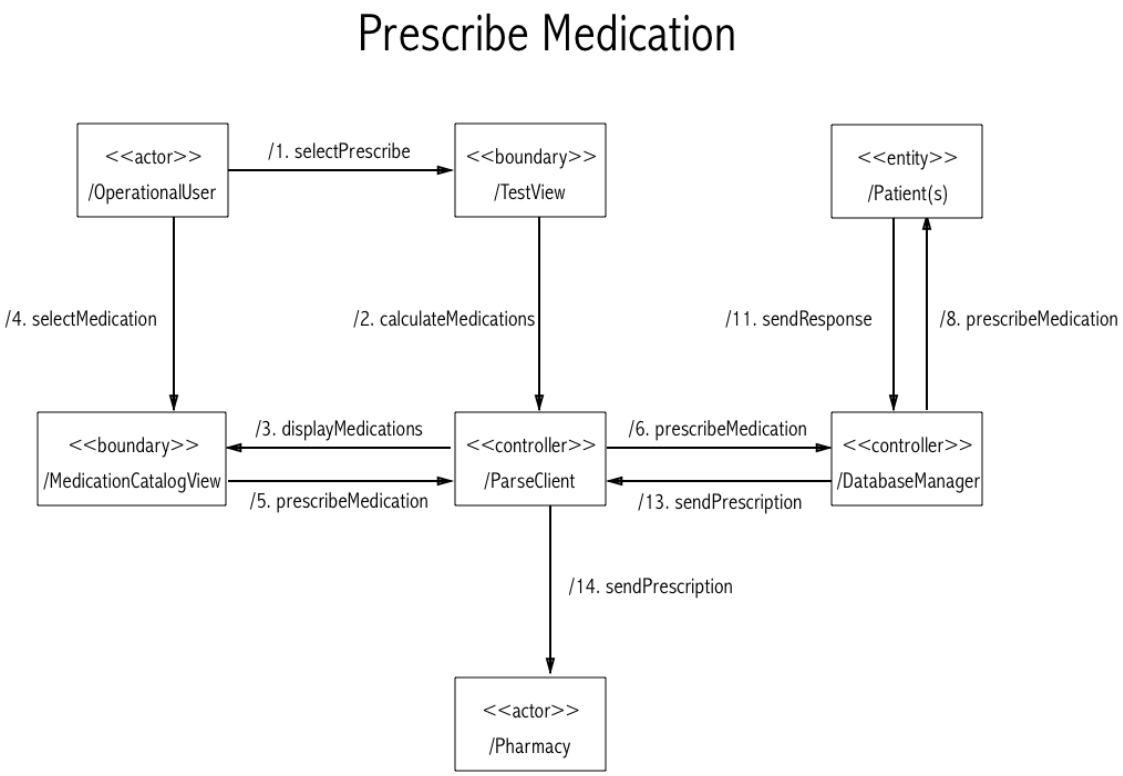
The Request Test use case mimic the GRASP patterns: Controller, High Cohesion.
ParseClient is a controller pattern by displaying tests to the TestCatalogView user interface.

Finish Test



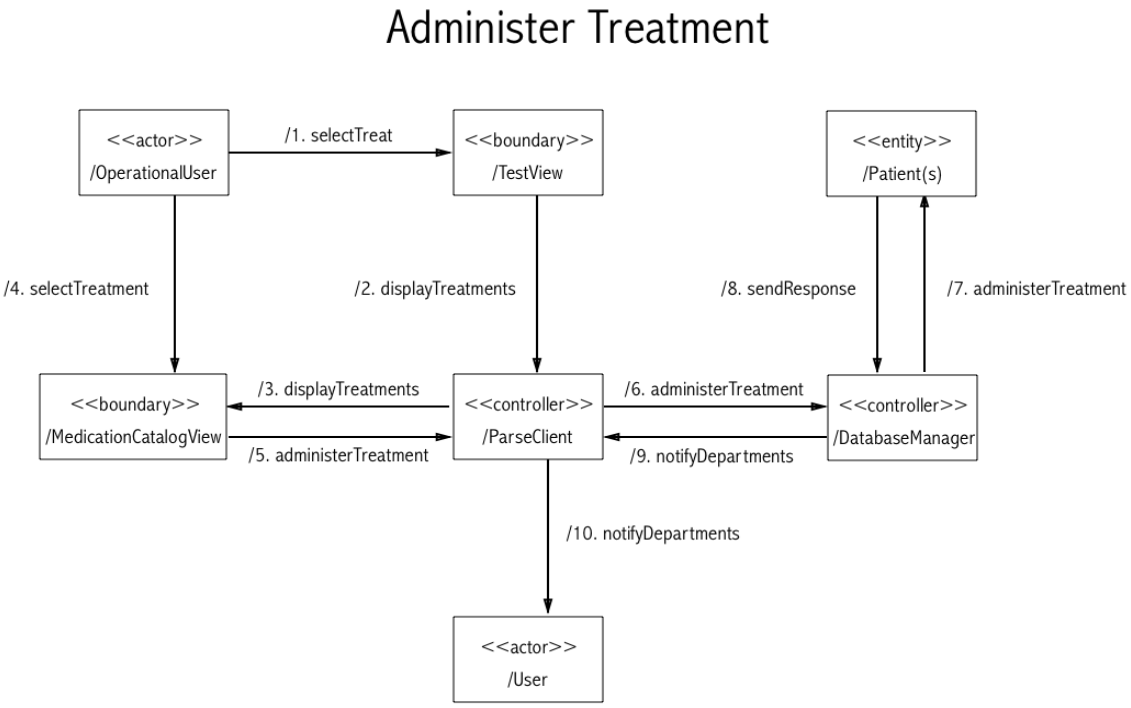
The Finish Test use case mimics the GRASP patterns: controller, high cohesion. The ParseClient controller has the responsibility sendFinishNotification which prompts the OperationalUser on the User Interface. The SendFinishNotification function causes high cohesion between object interactions.

Prescribe Medication



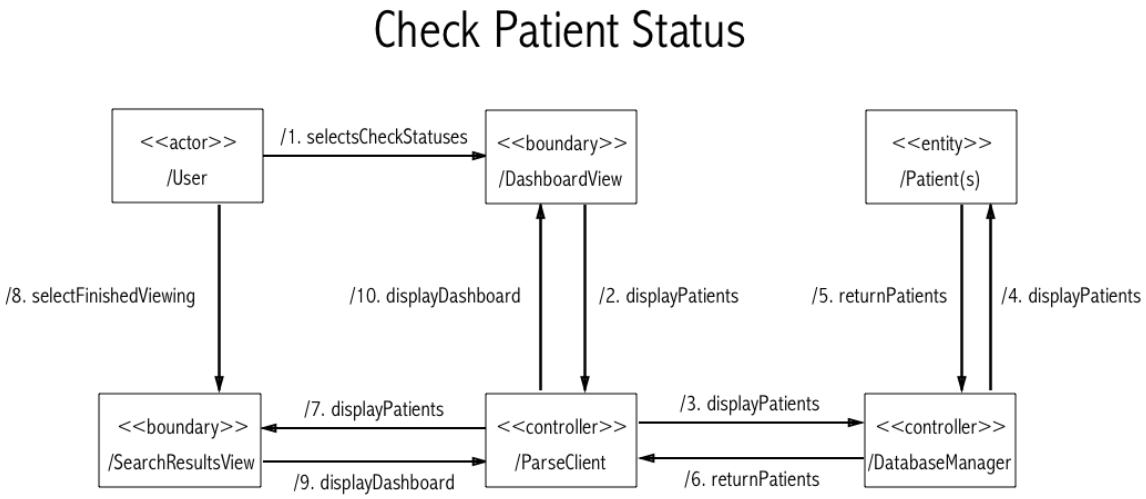
The Prescribe Medication use case mimic the GRASP patterns: Controller, high cohesion. The ParseClient displays Medication to the user interface MedicationCatalogView. Function such as, sendPrescription and prescribeMedication keep high cohesion in this use case.

Administer Treatment



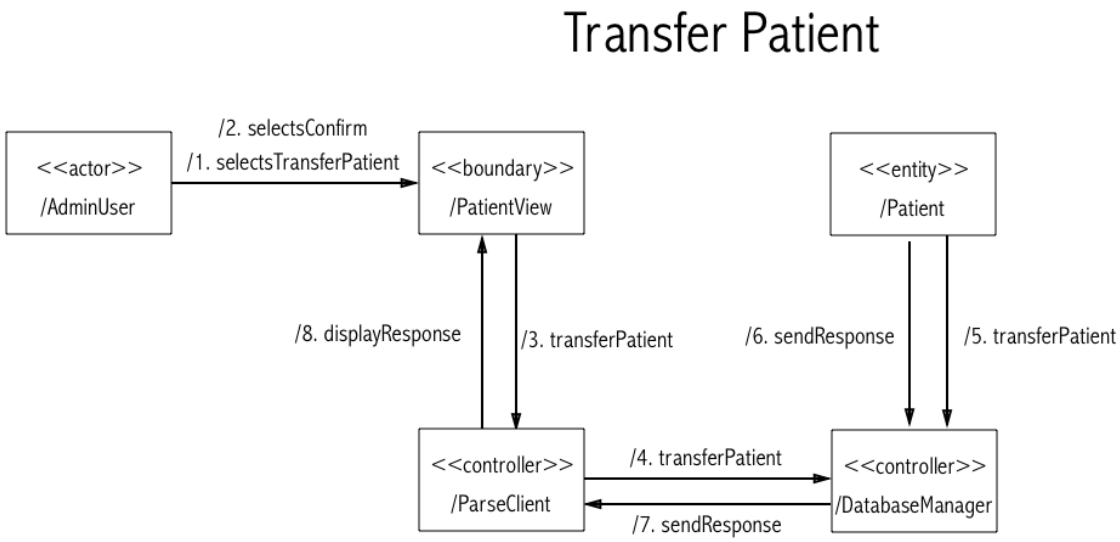
Administer treatment mimics the GRASP pattern(s): controller, high cohesion. Our interaction diagrams consist of the ParseClient controller which handles request for data on the user interface layer. The ParseClient handles operations from MedicationCatalogView which is displayed to the Operational User. As for the high cohesion, the entire class share similar responsibilities such as, AdministerTreatment and notifyDepartments.

Check Patient Status



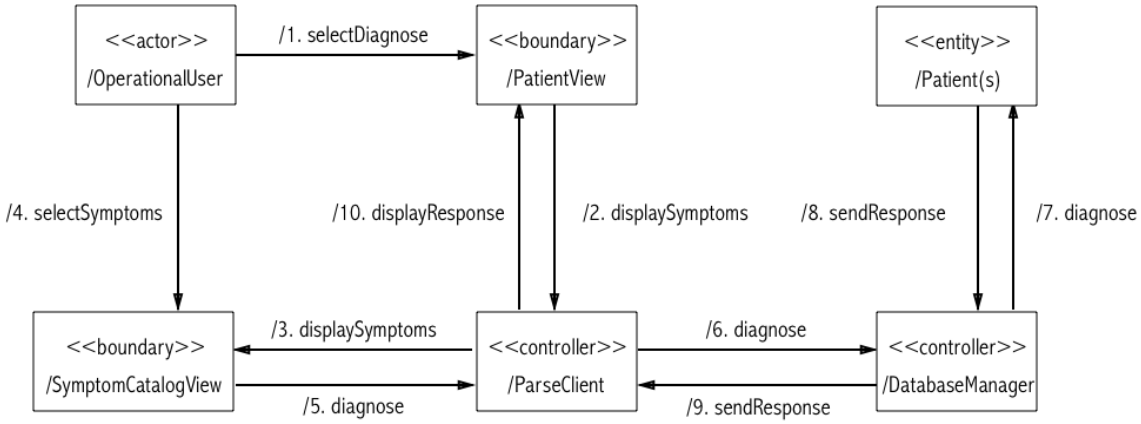
The Check Patient Status use case mimics the GRASP patterns: Controller, High Cohesion. ParseClient is a controller in this use case. It handles displaying patient to the SearchResultView user interface along with the DataBaseManager. The displayPatient function keeps high cohesion between object interactions.

Transfer Patient



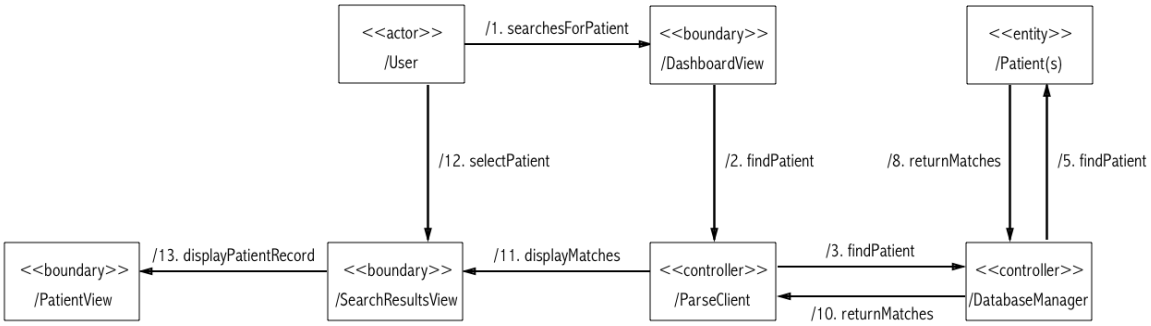
The Transfer Patient use case mimic GRASP patterns: Controller, High cohesion. ParseClient controller displays responses to the PatientView user interface. The transferPatient creates high cohesion between the object interactions.

Diagnose Symptom



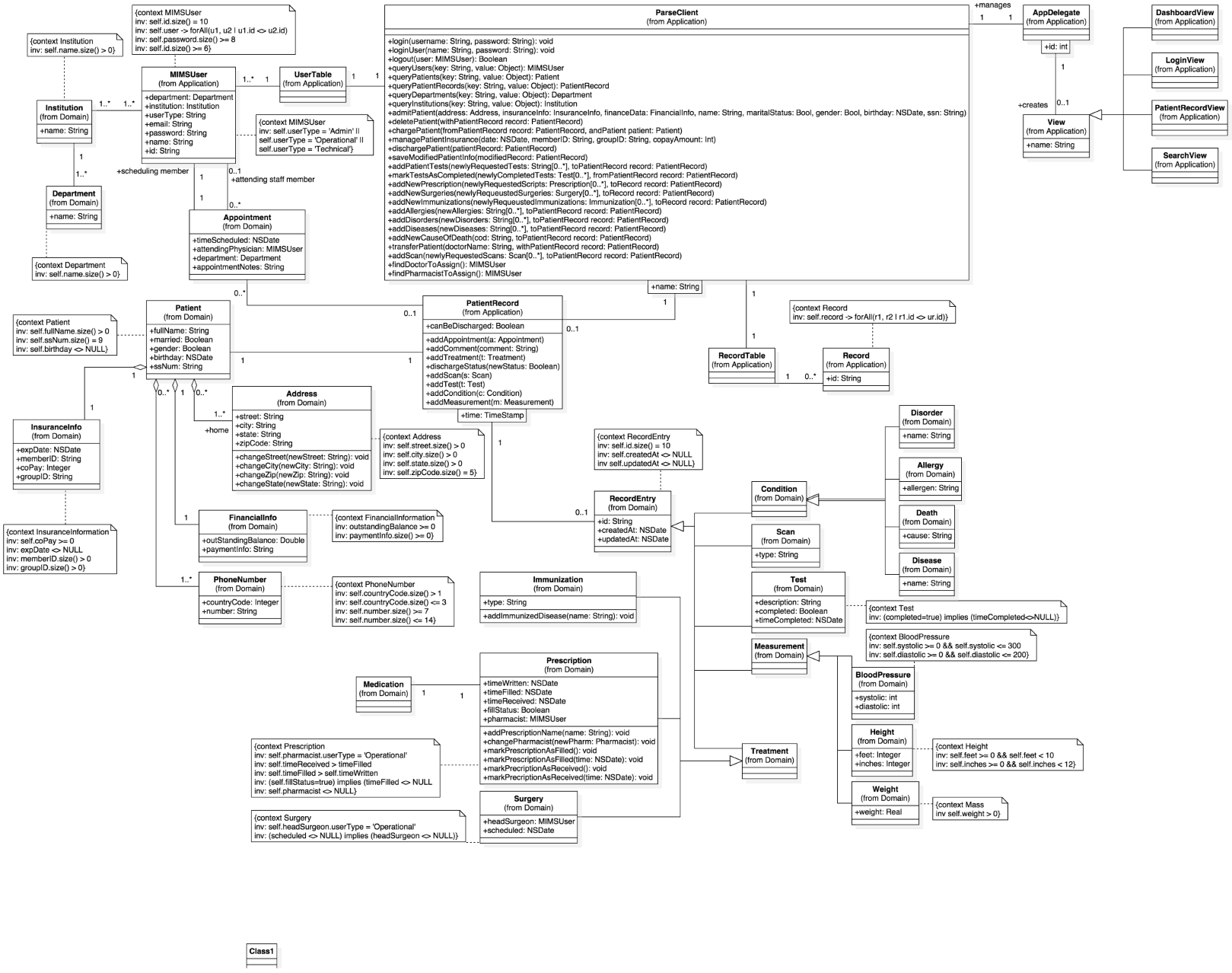
Diagnose Illness mimics the GRASP patterns: controller, high cohesion. In the diagnose illness use case, we designed three controllers ParseClient and DatabaseManager. Both the ServerController and DatabaseManager has the responsibility of diagnosing to the entity patient. The ParseController displays symptoms to the SymptomCatalogView user interface.

Load Patient Record



Load Patient Record use case mimics GRASP patterns: Controller, High Cohesion, Low Coupling, and Pure fabrication. The ParseClient controller handles the SearchResultView user interface. The use case has high cohesion such as the findPatient function which is consistently used during several object communications.

5.3 Design Class Diagram

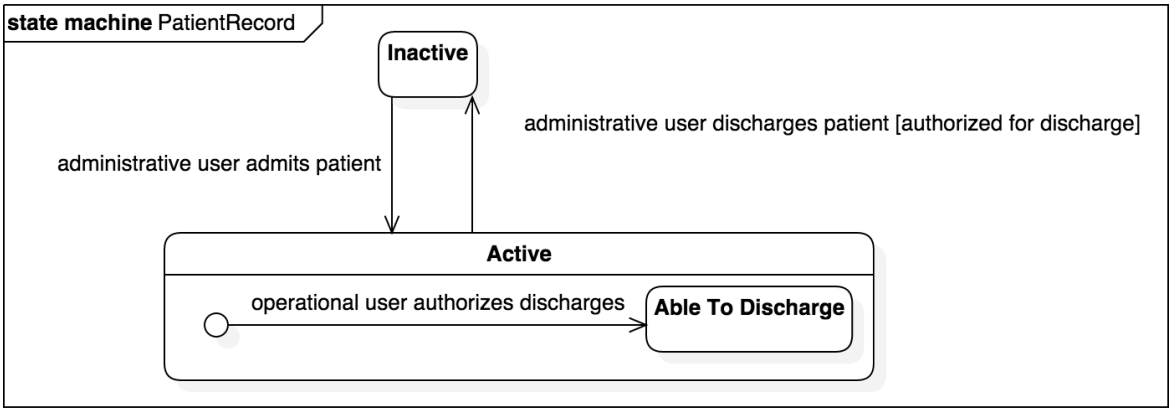


5.4 Object Design

Patient Record

a. Methods:
    i. addAppointment(newAppointment)
        1. Precondition: The PatientRecord object must exist already;
        2. Postcondition: The new appointment has been added to the PatientRecord
    ii. addComment(newComment)
        1. Precondition: The PatientRecord object must exist already
        2. Postcondition: The new comment has been added to the PatientRecord
    iii. changeActiveStatus(newStatus)
        1. Precondition: The PatientRecord object must exist already; the active status is not false already
        2. Postcondition: Set the active status to false
    iv. addScan(newScan)
        1. Precondition: The PatientRecord object must exist already
        2. Postcondition: The new scan has been added to the PatientRecord
    v. addTest(newTest)
        1. Precondition: The PatientRecord object must exist already
        2. Postcondition: The new test has been handed to the PatientRecord
b. Procedural Behavioral Specification
    i. Pseudocode for addAppointment()

```
addAppointment(newAppointment: Appointment)
    self.appointments.append(newAppointment)
```

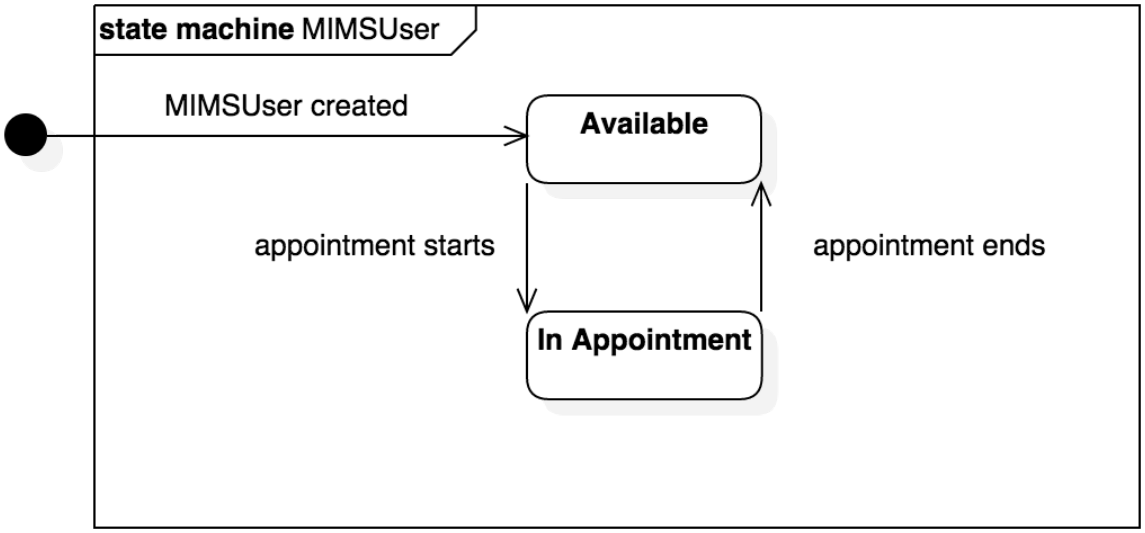


MIMSUser

a. Methods:
    i. Init(withUserType, department, institution)
        1. Precondition: The new MIMSUser is a user that is able to be created and given authentication to the system. The department and institution names are valid within the bounds of the MIMSUser class.
        2. Postcondition: The new MIMSUser has been created with a specific user type as well as an associated department and institution

b. Procedural Behavioral Specification
    i. Pseudocode for init()

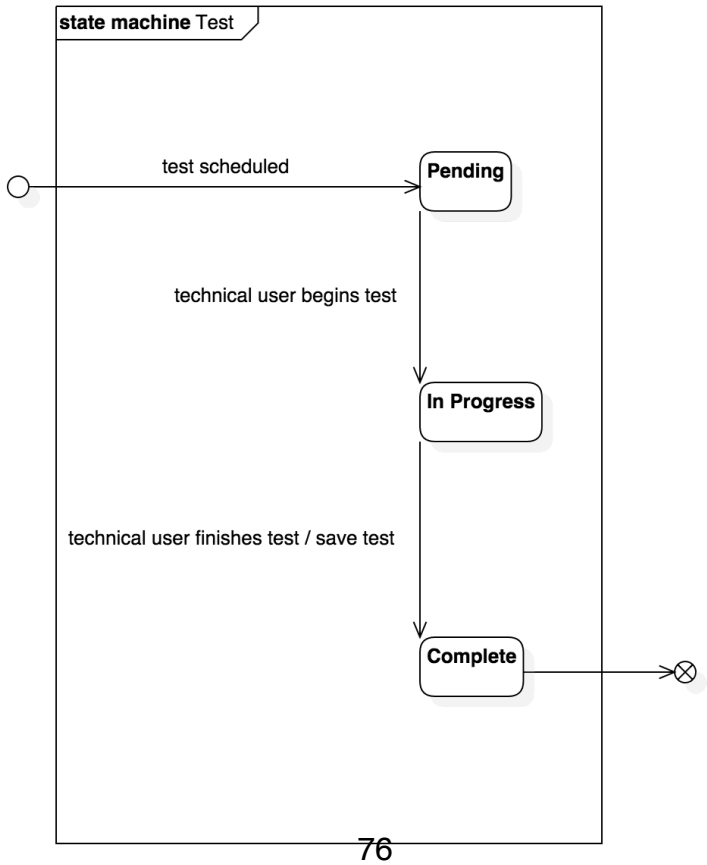
```
Init(withUserType type: String, departmentName:
String, institution: String)
guard let userType = UserTypes(rawValue: type) else
{ return }
self.userType = userType
self.department = Department(withName:
departmentName)
self.institution = Institution(withName:
institution)
self.saveEventually()
```



Test

a. Methods:
    i. Init(withTestDescription description: String)
        1. Precondition: The description string is a valid test name; There is a patient to add the test to when completed
        2. Postcondition: The test has been created. Its time assigned has been set, its completed status is false
    ii. changeCompletionStatus(newStatus, timeCompleted)
        1. Precondition: The test already exists and its completed status is false.
        2. Postcondition: Completed status is set to true, and the time completed has been assigned
    iii. markAsStarted(startTime)
        1. Precondition: The test already exists and its completed status is false.
        2. Postcondition: The time started has been assigned to the new time.
b. Procedural Behavioral Specification
    i. Pseudocode for markAsStarted()

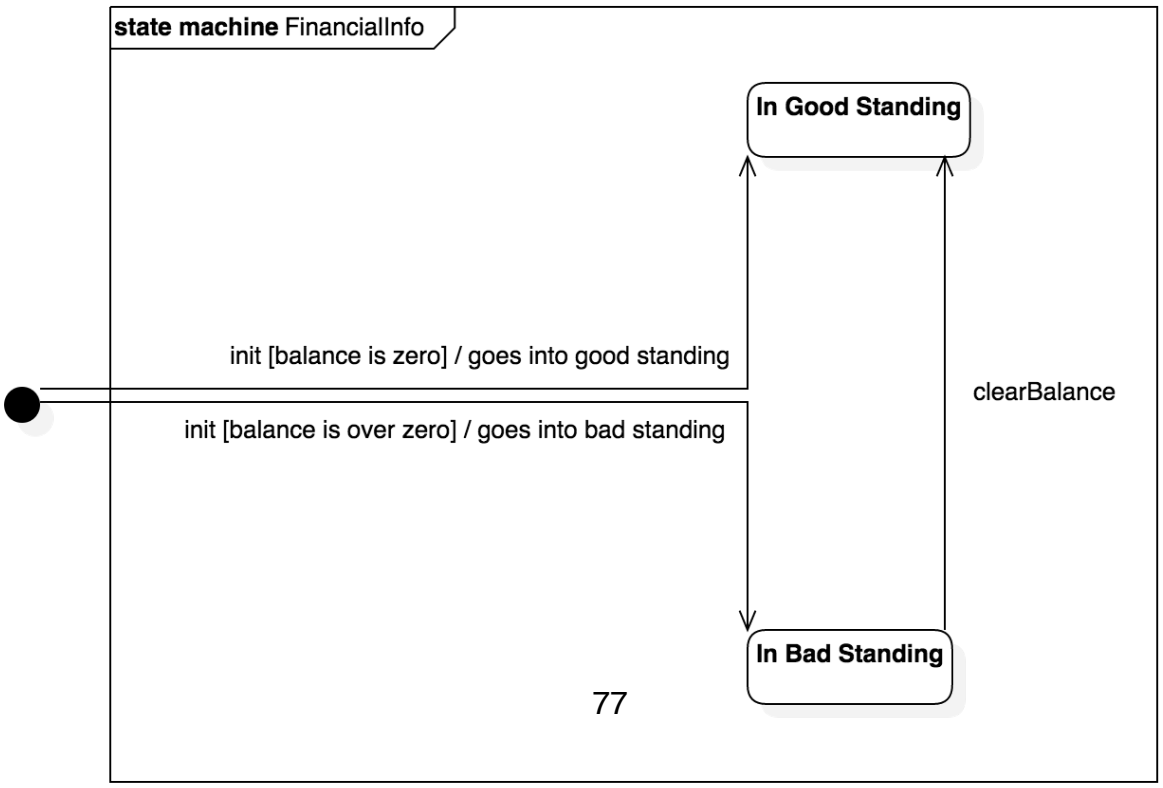
```
markAsStarted(startTime: NSDate)
if !completedStatus { self.startTime = startTime;
return true }
return false
```



FinancialInfo

a. Methods:
    i. Init(withPaymentInfo paymentInfo: String)
        1. Precondition: Payment info is not an empty string. There is a patient to add the new FinanceData to
        2. Postcondition: FinanceData has been created with the corresponding PaymentInfo. goodStanding has been set to true
    ii. Init(withPaymentInfo paymentInfo: String, newBalance: Double)
        1. Precondition: Payment info is not an empty string; there is a patient to add the new FinanceData to; the new balance is greater than 0
        2. Postcondition: FinanceData has been created with corresponding parameters. If the new balance is greater than 0, set goodStanding to false, otherwise it's true
    iii.clearBalance()
        1. Precondition: The FinanceData has an outstanding balance.
        2. Postcondition: The FinanceData good standing has been set to true. The balance has been set to 0.
    iv. inGoodStanding()
        1. Precondition: The FinanceData exists for the certain patient.
        2. Postcondition: N/A (no changes)
b. Procedural Behavioral Specification
    i. Pseudocode for inGoodStanding()

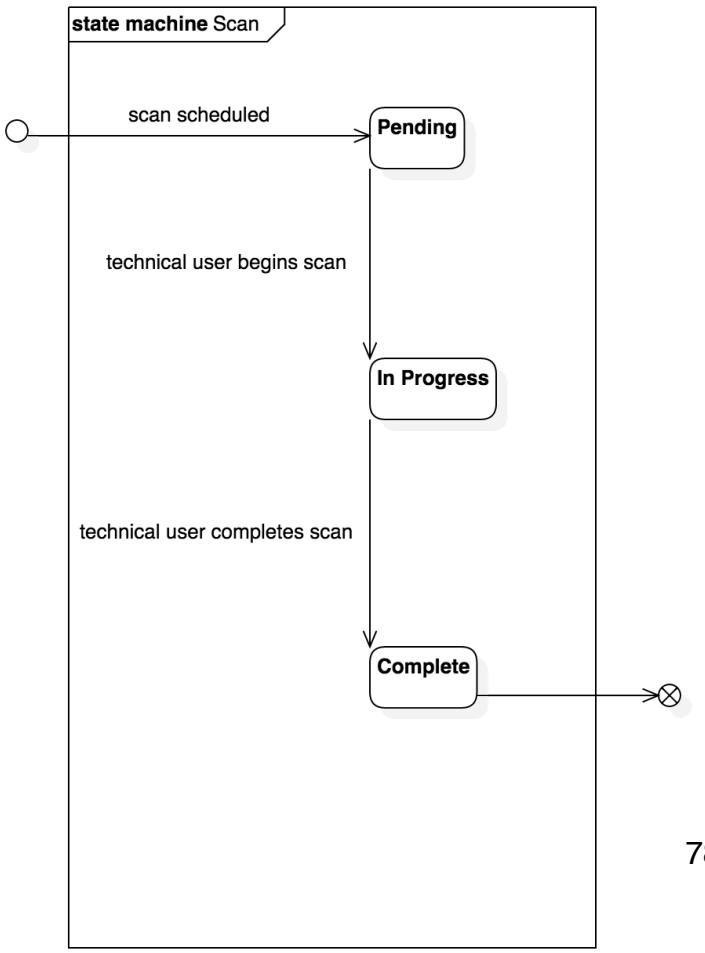
```
inGoodStanding()
return goodStanding!
```



Scan

a. Methods:
    i. Init(withScanDescription description: String)
        1. Precondition: The description string is a valid scan name; There is a patient to add the scan to when completed
        2. Postcondition: The scan has been created. Its time assigned has been set, its completed status is false
    ii. changeCompletionStatus(newStatus, timeCompleted)
        1. Precondition: The scan already exists and its completed status is false.
        2. Postcondition: Completed status is set to true, and the time completed has been assigned
    iii. markAsStarted(startTime)
        1. Precondition: The scan already exists and its completed status is false.
        2. Postcondition: The time started has been assigned to the new time.
b. Procedural Behavioral Specification
    i. Pseudocode for markAsStarted()

```
markAsStarted(startTime: NSDate)
if !completedStatus { self.startTime = startTime;
return true }
return false
```



6 Evaluation

The Medical Information Management System is best described as a client-server relationship, with MVC overtones. The client-subsystem is based on a multi-page application where each page can manage certain requests, and this is very common with mobile applications. Specifically, with mobile applications you don't want to try and accomplish too many features into a single page because it's easy for the application to become confusing and overbearing very quickly. In order to assist with this implementation, we used a server manager known as Parse which takes the guesswork out of managing your own server, and instead provides a large API that enables us to reach our goals in an easy but sustainable manner. The server subsystem uses a database, comprised of many tables, to store and access the data used by the system. Finally, the database and associated interface are managed by the Parse library. Thus, logically it follows that the system can be broken up into three discrete subsystems: Client, Server, and Database.

This detailed design model for MIMS is best evaluated by individually evaluating the completeness, consistency, and design quality of each individual component, as well as the detailed design as a whole.

Overall, MIMS has a medium level of completeness. Of course there is always room for additional completeness and MIMS is no different. However, the system very much addresses the specified used cases to a high degree of completeness. Additionally, the system's modular architecture allows for high maintainability and creates a somewhat clear path forward for additional features. The system modeling is mostly complete. There is almost no way to get an entire graph of all the modeling that needs to be done in order to completely design a system, and once you begin implementation, it's easy to see where things have gone wrong in the design process. However, with what we have, the UML diagrams provide a clear picture of the structure of our system, and as far as we can tell there are no glaring faults that hinder the structure of our models. For instance, cardinality is set for whenever associations are necessary and all attributes for a class have unique names with several constraints indicating what can and cannot be valid values for a given attribute. Additionally, there are no unused design artifacts, and every component within individual diagrams and throughout the model has a specific purpose.

The consistency of MIMS is relatively high, but could use work. We've spent a lot of time trying to make sure there is consistency between class objects and their corresponding system sequence diagrams, but while working as a group, everyone has different naming conventions and some of these inconsistency issues may still lie within the design. That being said, there is a lot of consistency throughout the models. For instance, the objects in each system sequence diagram can be found in the class diagrams with the exception of a specific class like DatabaseManager because our implementation handles database management for us. The Use Case diagrams and Sequence Diagrams are consistent with each other for all the given use cases because for each use case, the corresponding Sequence Diagrams correctly models the Use case. The Sequence diagrams and State Diagrams are consistent with each other, as most but not all state transitions for an object are shown in the sequence diagram by the

messages sent and received by the object. Additionally, some of the Statecharts used in the model show internal consistency through the fact that in each state there are few events that can allow a transition.

MIMS the system itself is also high quality. A system's level of quality is based on its reliability, efficiency, maintainability, and usability. MIMS certainly possesses these traits.

For starters, the system is very reliable. The system delivers consistent and reliable data in response to user requests. (One caveat of the system is that on first access, the server needs time to "start up" because it runs in a mode that allows it to sleep for efficiency when not being used.) Furthermore, system results are repeatable, indicating high consistency. The system also handles all possible inputs from the user, so it has a high level of completeness, and doesn't leave some inputs up for grabs in terms of who or what should be responsible for it. A server-reliant application is certainly subject to limitations however, and some requests will result in exception cases. These exceptions stem from the fact that a system such as this is not self-contained, as it must communicate across servers. Despite this, even errors in network connectivity are anticipated by the system and will lead to expected results, such as a time out or an error message if something is unable to be finished. This level of anticipation indicates a system that is reliable and highly robust.

MIMS is also highly efficient. While efficiency is hard to measure sometimes, it can be reasonably assumed that the simple client-server and MVC architecture will lead to a high level of efficiency, especially considering the previously mentioned reliability. For instance, the system will limit requests to maintain low levels of bandwidth usage, and will also cache the data that it receives so that the request can be executed locally in the cache before reaching out to the server for new information. Another system might've required several queries to accomplish retrieving the same information that we can retrieve in one, or none if that values have already been cached. This is highly lucrative for us, especially in mobile implementation, as the amount of data used and time spent computing are eternally valuable.

Maintainability is an important measure of quality as well. In MIMS, maintainability isn't quite as great as it could be because sometimes an architecture such as ours requires some level of interconnectedness. For instance, the URL of the database is encoded in the ClientController, and a change in the location of the database would require a new value to be associated with here. However, despite this limiting factor, the individual subsystems still have a high degree of maintainability. For instance, the client subsystem is built upon reusable modules for getting and setting data on the database, and are encapsulated and self-contained. The intelligent separation of concerns has lead to a client-server system that is maintainable, despite some obvious limitations.

Finally, MIMS is quite usable. From a UX perspective, the system is approachable and easy to use. The user interface is consistent in its placement and styling of buttons, lists, forms, and other design elements, which results in a cohesive experience for the common user. Furthermore, extraneous features have been removed to keep the design clean and approachable to the user. For any type of account, the dashboard features a subset of easily viewable things, easily accessible buttons to perform expected operations, and an at-a-glance view. From the dashboard, it's

possible to do almost anything a user would expect to be able to do with minimal navigation. From a design standpoint, the system is modern and attractive, and more interactive than typical software you might find in other medical management systems. This might seem like a minor point, but studies commonly demonstrate that the use of good design principles results in lower cognitive friction for users and results in a better user experience. Of course, the actual usability of the system is directly related to the user experience.

Using OO design quality metrics, the system-wide Average Method Complexity is incredibly low. The vast majority of methods are simply setters and getters. Of course there are more complex methods that incorporate multiple simple steps, but the resulting AMC is still quite low. Most functions take up less than 15 lines of code and are capable of handling their functionality in quick and efficient ways. The system-wide Percentage Public and Protected (PAP) results in a good mix of private variables that back some of the publicly accessible variables.

As the previous paragraphs have exhaustively explained. MIMS is well-designed. The client-server and MVC architecture is appropriate and has resulted in a system that has high completeness, consistency, and design quality.