

Design Document for:

PatientNow!

Written by
Johnathan Nicholson
Seth Barrios
Paula Ledgerwood
Christiana Ushana
Jackson Darrow
David McIntyre

Version # 1.00

Nov. 19, 2016

| | |
|---|-----------|
| PatientNow! | 1 |
| 1.0 Introduction | 5 |
| 1.1 Goals and objectives | 5 |
| 1.2 Statement of scope | 5 |
| 1.3 Software context | 6 |
| 2.0 Data design | 7 |
| 2.1 Internal software data structure | 7 |
| 2.2 Global data structure | 7 |
| 2.3 Temporary data structure | 7 |
| 2.4 Database description | 7 |
| 3.0 Architectural and component-level design | 8 |
| 3.1 System Structure | 8 |
| 3.1.1 Class diagram | 8 |
| 3.2 User | 9 |
| 3.3 Appointment | 11 |
| 3.4 Room | 13 |
| 3.5 RoomType | 14 |
| 3.6 Employee | 15 |
| 3.7 Medical Professional | 17 |
| 3.8 Shift | 19 |
| 3.9 Patient | 21 |
| 3.10 Conversation | 23 |
| 3.11 StaticResourceManager | 25 |
| 3.12 AppointmentManager | 27 |
| 4.0 User interface design | 28 |
| 4.1 Description of the user interface | 28 |
| 4.2 Interface design rules | 36 |
| 4.3 Components available | 37 |
| 5.0 Appendices | 38 |
| 5.1 Packaging and installation issues | 38 |
| 5.2 Diagrams | 38 |
| 5.2.1 Class Diagram | 39 |
| 5.2.2 Activity Diagrams | 40 |

Revision History[Gautam S.1]

| Name | Date | Reason For Changes | Version |
|------|------|--------------------|---------|
| | | | |
| | | | |

1.0 Introduction

The purpose of this document is to give an overview of the PatientNow! system providing an insight into the structure and design of each part of the software. Topics covered will include the following.

- Class hierarchies and interactions
- Data flow and design
- Processing narratives
- Algorithmic models
- Design constraints and restrictions
- User interface design
- Test cases and expected results

By the end of this document the reader should have an understanding of how PatientNow! works.

1.1 Goals and objectives

The purpose of PatientNow! is to provide medical practices with an easy way to host their own website packaged with a scheduling system and care plan management system. The primary objective of the PatientNow! project is to provide an easy to use alternative to medical practices developing their own websites.

In order to reach these goals the finished product must have all the traits of a successful web application including a visually appealing user interface and a responsive easy to use user experience.

1.2 Statement of scope

The PatientNow! System is composed of two primary components: a client-side website which recieved user input and a server-side application which updates and synchronizes data across the website. The system features can be broken up into two groups as well: core features, which are essential to the function of the website, and additional features, which are only meant to add extra functionality. The following list includes all of the features currently designated for inclusion in the final release of PatientNow!:

Core Features:

1. User Registration & Welcome
 - Allows user to register with PatientNow! Server
 - Enables the user to edit their account settings and preferences
2. View and Edit CarePlan
 - Patient will receive push notification when CarePlan has been modified
 - Patient will be able to view CarePlan
 - Medical Professional will be able to view and edit CarePlan
3. Charge Patient For Service Provided
 - Practice can send patient a bill to pay through third party billing system
4. View Bill
 - Patient will be able to view bill
 - Patient will pay through third party system
5. Modify & View Schedule
 - Medical professionals & Administrative Assistants will be able to modify their availability
6. Modify & View appointments
 - Administrative Assistant will be able to modify type of room to appointment
 - Administrative Assistant will be able to modify type of appointment
 - All users will be able to view all appointments
7. Edit Basic Patient Information
 - User will be able to change their password
 - User will also be able to change contact information
8. Communication with Patient
 - Doctor can create unlimited message threads with patient
 - Doctor can send messages to the patient
 - Doctor can choose to make the conversation replyable

Additional Features:

1. Help Menu
 - Displays a list of options covering the different components of PatientNow!
 - Offers information on each feature, menu, etc
 - Can be accessed at any time via the Settings menu
2. Settings Menu
 - Allows the user to customize his/her preferences
 - Enables the user to modify certain features and functionalities
 - Can be accessed at any time on every page of website

1.3 Software context

PatientNow! Will be a web application designed to provide an easy way for medical practices to streamline a number of office workflows. Once the initial development of the website is complete, medical practices will be able to host customized instances of our website at virtually no cost to us.

Future development will be determined by consumer demand for new features.

2.0 Data design

Most of the Data will be stored representation of the classes listed in the attached UML Class Diagram (see the appendix).

2.1 Internal software data structure

Many of the data objects will be represented as classes with properties defined in the UML Class diagrams. Where collections of objects will be generally stored as unsorted lists since there is not a natural way to sort them considering that they could overlap in time.

2.2 Global data structure

A List of Users will be available to a majority of the application as users start most of the actions associated with this system.

2.3 Temporary data structure

We will have temporary lists of calculated information such as Medical Professionals available at a given time or for conflicting appointments when an employee's shift has been changed. We will also have temporary JSON objects that are sent between our Web based client and our server.

2.4 Database description

As the database is MySQL 5.7 the data will be defined in a relational manner, the relationships defined in the class diagrams as they relate to the following classes.

- User
- MedicalProfessional
- Employee
- Room
- RoomType
- Appointment

- AppointmentType
- Patient
- Conversation
- Message
- File
- Staffing
- Shift

3.0 Architectural and component-level design

3.1 System Structure

The PatientNow! system is broken up into two major components a client side javascript application and a server side Java (v 1.8) and MySQL (v 5.8) application.

The client-side application will be a light interface mostly dedicated to displaying the information provided by the server in a clear and consistent manner. This means that the client-side application will also not have much of the business logic.

The server-side application will serve to process requests sent by the clientside and to persist data via the MySQL instance. This server will process requests via a Tomcat servlet with Java logic and a hibernate ORM connection to the database. These communications will be applied to the database in transactions to avoid parallel requests causing erroneous data to be reported to the user, such as double booking a room.

3.1.1 Class diagram

The class diagram is located in the appendix to save space. See section 5.2.1.

3.2 User

The User class represents people who use the system. Each person who uses the system is associated with one user object, and this user instance keeps track of both account information (email, name, password, etc.) and roles that the user has within the system. Based on the roles associated with a user, the system can determine what functionality is available to that person.

3.2.1 Processing narrative (PSPEC)

When a user first creates his/her account they are assumed to be a patient unless another account type is specified on creation. Each time that the user begins a new session with this software, this User object is referenced. This object is responsible for storing the basic information tied to any user (Patient, Assistant or Medical Professional) including

- id : Integer
- username: string
- passwordHash: string
- firstName: string
- lastName: string
- email: string

3.2.2 Interface

- isPatient() : boolean
- isProfessional() : boolean
- isAssistant() : boolean
- sendNotification() : void
- checkPassword(pass) : boolean

3.2.3 Processing Detail

This class supports the following functionality:

1. **Authenticate the user:** Once the system has retrieved a password from the user, checkPassword(pass) is called in order to determine whether the User may further interact with the system.
2. **Record/modify basic user data:** The User class has a series of properties that can be set/retrieved via setter/getter calls. The setter of the 'firstName', 'lastName', 'password', and 'email' fields may be called when an authenticated User indicates his/her want to change these values via UI interactions. The 'username' setter should be called once at creation time of the User instance. The 'id' and 'passwordHash' fields should only be determined by the system (not by a User).
3. **Determine permissions within system:** isPatient(), isProfessional(), and isAssistant() are called when components of the system would like to determine what functionalities are available to a User instance.

3.2.3.1 Design Class Hierarchy

The User class does not inherit from any other class. Users have references to Role instances based on their position within the practice (patient, employee, medical professional).

3.2.3.2 Restrictions/Limitations

A User can have at most (1) reference to a Patient role and (1) reference to an Administrative Assistant **or** Medical Professional role. The system should ensure that no User breaks this rule.

3.2.3.3 Performance Issues

Hashing a password is the gate to logging into our system, this hashing process will need to be balanced based on a combination of security needs and server speed, however BCrypt with a strength value of 12 (about 2^{12} rounds) should be near this balance.

3.2.3.4 Design Constraints

As mentioned in the restriction portion (3.2.3.2), a User should reference a Medical Professional **or** an Employee role. Because the Medical Professional class inherits from Employee, having a reference to both is not logical.

3.3 Appointment

The appointment class is essential to this scheduling software. The system is expected to constantly be adding, modifying, and reading from appointments. Appointment objects that exist in the system reflect appointments that which physically take place in the medical office.

Specifically, the appointment class is used to 'link':

1. A patient
2. A block of time
3. A room
4. 1 or more medical professionals

3.3.1 Processing narrative (PSPEC)

An appointment is created for any reservation of time by a patient. Once created, a patient cannot modify the appointment except through cancellation. Several other events may cause an appointment to be altered, however. Staff of a medical office may modify the values listed above. Additionally, the adjustment of the work schedule (for instance, a medical professional calls in sick), may cause appointments to become invalid. Because appointments hold a relationship to the associated patient, both the medical office and the patient will be notified when a change has occurred. This object will live beyond its occurrence time in the database as a record of the event.

3.3.2 Interface

- `getStartTime()`
- `setStartTime()`
- `getEndTime()`
- `setEndTime()`
- `changeType()`
- `changeRoom()`
- `getPatient()`
- `setPatient()`
- `getStaff()`
- `setStaff(MedicalProfessional)`
- `notifyPatientOfChange(message)`
- `generateBill()`

3.3.3 Processing Detail

The User class primarily exists as a record for linking a Patient instance and resources of the practice. Here are the functionalities it supports:

1. **Link and retrieve resources/Patient:** Except upon creation of an Appointment instance, the setters for 'startTime', 'endTime', 'appointmentType', 'room', 'patient', and 'staff' should generally not be called. The getters can be called at any time and are

called when the system needs to display information about existing appointments to the user.

2. **Update a Patient about vital appt. change:** Whenever an Appointment instance is modified, notifyPatientOfChange(message) is triggered to update the Patient about those changes. This functionality is to be used when a shift is cancelled (Shift instance removed from the system) and corresponding Appointments are invalidated.
3. **Generate a bill:** A call to generateBill() must originate from an Administrative Assistant or Medical Professional instance. When called, a bill is generated, which can then be linked to a Patient instance for displaying a bill to the user.

3.3.3.1 Design Class Hierarchy

This class has an associated room and appointment type. It also has a list of medical professionals associated with this appointment and finally a patient.

3.3.3.2 Restrictions/Limitations

Because of the persistence of this class, the stored information should be kept to a minimum, therefore this class is mostly references to other pieces of data to both maintain normalized form and to decrease size per appointment.

3.3.3.3 Performance Issues

To avoid double booking rooms appointment creations must be done transactionally. This could present an issue should the amount of appointment requests get too large.

3.3.3.4 Design Constraints

Because appointments can have multiple medical professionals and vice-versa, Appointments have a list of Staffings which reference the associated professional.

3.4 Room

The Room class represents a room (available for medical purposes) physically located in the medical practice office. Each room can service a set of appointment types. During a given time period, a 'Room' instance can only be associated with one (or fewer) 'Appointment' instances.

3.4.1 Processing narrative (PSPEC)

Processing Narrative: Upon setup of our system the rooms of the building will be added as this Room class. These rooms serve to prevent both overlapping appointments in the same space and misallocation of rooms for purposes that they were not meant for.

3.4.2 Interface

- isAppointmentAllowed(Appointment) : boolean
- checkConflict(startTime, endTime): boolean

3.4.3 Processing Detail

This class is a resource and therefore is only responsible for representing a physical room of the practice. The following functionality is supported:

1. **Verify that an appointment is valid:** isAppointmentAllowed() is called to ensure that the appointmentType of the Appointment is allowed by the Room instance's roomType. In addition, checkConflict() verifies that no Appointment with conflicting times is already scheduled for that Room. These functions are called when generating a new Appointment instance.

3.4.3.1 Design Class Hierarchy

This class has a list of allowed appointment types and is referenced by specific appointments.

3.4.3.2 Restrictions/Limitations

This class must carefully run the test for overlapping use of the room in a transactional manner to avoid rooms being double booked.

3.4.3.3 Performance Issues

N/A

3.4.3.4 Design Constraints

N/A

3.5 RoomType

This class is meant to define the various types of rooms that can exist within a medical space. These types define the rooms that exist so as to be clear the functionality of each room and make setup easier.

3.5.1 Processing narrative (PSPEC)

This class is created upon the setup of the practice and exists with minimal changes as a reference to what rooms can handle what appointments.

3.5.2 Interface

- `isAllowed(AppointmentType) : boolean`
- `getName() : string`
- `setName(name:string)`
- `addAllowed(AppointmentType): void`

3.5.3 Processing Detail

This class is responsible for linking rooms to allowed appointment types without forcing the practice to specify all appointment types each time. The following functionality is supported:

1. **Allow a RoomType to service an additional appointmentType:** When a practice decides that a given RoomType can service a new appointmentType, `addAllowed(AppointmentType)` is called on an existing RoomType instance.
2. **Verify an AppointmentType is allowed for a given RoomType:** `isAllowed(AppointmentType)` simply returns whether a RoomType's can service a given AppointmentType. This is called when generating a new Appointment instance (specifically when finding Room instances that are valid for an AppointmentType).

3.5.3.1 Design Class Hierarchy

Rooms have a single room type and room types have a list of allowed appointment types.

3.5.3.2 Restrictions/Limitations

N/A

3.5.3.3 Performance Issues

The operations performed by this class are not computationally expensive, nor are they called exceedingly often.

3.5.3.4 Design Constraints

This class was constructed to ease the setup of the practice.

3.6 Employee

This is the general form of employees for the practice. This exists in order to provide a method of modifying shifts for employees while also allowing employees and medical professionals to create appointments. The employee class can also check for appointments that conflict with a shift change using the medical professional's ability to get shifts. If the employee is not a professional then no shifts should be affected.

3.6.1 Processing narrative (PSPEC)

A new 'Employee' is created when a new administrative assistant is hired by the practice. In order for the an 'Employee' instance to be useful to the system, it must have a list of shifts associated with it--this list represents the work schedule of the employee, during which he/she will be working. The Employee class is responsible for maintaining shifts and creating new appointments. Shifts can be added to or removed from the Employee instance at any time during its existence.

3.6.2 Interface

- addShift()
- removeShift()
- createAppointment(Prof, Patient)
- getInvalidatedAppointments(): List<Appointment>

3.6.3 Processing Detail

This class supports the following functionality:

1. **Maintain shifts:** When an employee's work schedule changes, he/she updates the system to reflect changes via addShift() and removeShift(). These functionalities will primarily be triggered via UI events.
2. **Generate appointments:** createAppointment(Prof, Patient) is called when an employee schedules an appointment on behalf of a patient. This function explicitly links a Patient and Medical Professional instance with an Appointment instance.
3. **View invalidated appointments:** If new invalidated appointments exist in the system (due to the removal of a Shift instance from a Medical Professional), getInvalidatedAppointments() is called so that an Administrative Assistant can decide how to handle those invalidations (e.g. contacting an affected patient to schedule a new appointment).

3.6.3.1 Design Class Hierarchy

This class is extended by Medical Professional. This class also has a list of shifts that are associated with that user.

3.6.3.2 Restrictions/Limitations

If a user is a medical professional they should not have both an employee role and a medical professional role.

3.6.3.3 Performance Issues

Checking for shift conflicts could be a database intensive task, however this is deferred to the Medical Professional class

3.6.3.4 Design Constraints

While this class does not directly have appointments and conversations, it does have shifts. Because medical professionals extend this functionality, there must be a way to check for conflicts through this class.

3.7 Medical Professional

This class is a role for user that allows for the creation of care plans, the creation of appointments and includes all of the functionality that an employee is afforded. In addition the medical Professional class can get appointments that are within a certain startTime and endTime, should a shift change.

3.7.1 Processing narrative (PSPEC)

This class is a role for user that allows for the creation of care plans, the creation of appointments and includes all of the functionality that an employee is afforded. In addition the medical Professional class can get appointments that are within a certain timeFrame, should a shift change.

3.7.1 Processing narrative (PSPEC)

Medical Professional is a class that maintains a link between a user that is a medical professional and the various appointments and conversations they may have. This object is created when a medical professional is registered and is referenced whenever shifts, or appointments are changed related to this medical professional. This class also manages creating and managing Conversations between the doctor and the patient. This class also is responsible for facilitating the collection of conflicting appointments should a shift change.

3.7.2 Interface

- createConversation()
- getAppointments(startTime, endTime)
- getAllAppointments()

3.7.3 Processing Detail

This class supports the following functionality:

1. **Start new conversations with patients:** In order to communicate information regarding a patient's care plan, a Medical Professional instance can generate a new Conversation object via createConversation(). This new Conversation instance is linked to a Patient instance, so that both Users can begin posting messages to it.
2. **Retrieve list of appointments:** When a medical professional wants to view his/her upcoming appointments, getAllAppointments() or getAppointments(startTime, endTime) is triggered on the MedicalProfessional. The returned list of Appointments can then be filtered and displayed (via UI) to the corresponding user.

3.7.3.1 Design Class Hierarchy

This class extends the Employee class and therefore has a link to all the information that Employees have. In addition, Medical Professionals have a list of Appointments in the form of Staffings.

3.7.3.2 Restrictions/Limitations

N/A

3.7.3.3 Performance Issues

N/A

3.7.3.4 Design Constraints

The Medical Professional class **does not** inherit from User. Rather, a User instance references a single Medical Professional instance. While it may seem more intuitive for the Medical Professional class to inherit from User, this configuration allows a User to be both a medical professional and patient at the practice.

3.8 Shift

A shift represents the working hours of an employee. This class also has the ability to check for appointments that would be affected by a change to this shift.

3.8.1 Processing narrative (PSPEC)

A new instance of Shift is created when an employee indicates that he/she is going to work during a specific period of time (specific meaning that the period occurs on a specific date and has a start and end time). A new Shift is added to an Employee instance's list of Shifts. The system can then use Shift Instances to determine if a Medical Professional (a subclass of Employee) can be scheduled for appointments. If a Shift must be canceled (removed), the system should retrieve its list of associated appointments and broadcast the invalidation of those appointments.

3.8.2 Interface

- addAssociatedAppointment(Appointment): void
- removeAssociatedAppointment(Appointment): void
- getAssociatedAppointments(): <List> Appointments

3.8.3 Processing Detail

A shift has two main functionalities that it supports.

1. **Maintain start/end time:** an instance of shift records a start and end time of the shift. These two properties should be set when a Shift instance is first created. They should also be retrievable throughout the lifetime of the instance.
2. **Retrieve associated appointments:** A Shift instance should be able to return a list of appointments occurring during that shift. For instance, a medical professional may have a shift--during that shift, he/she attends a number of appointments. A Shift instance for that medical professional returns that list of appointments. This functionality is represented in the interface by getAssociatedAppointments(). Modifying the list of associated appointments can be done via the add/remove functionalities shown in the interface above.

3.8.3.1 Design Class Hierarchy

This class does not inherit from any others. Both the Employees and Medical Professionals have a list of Shifts, though only the Medical Professional can be scheduled for appointments.

3.8.3.2 Restrictions/Limitations

A shift's start and end times must be kept within the working hours of the practice.

3.8.3.3 Performance Issues

There is potentially room for inefficiency when retrieving the list of associated appointments. The shift should not need to iterate over all appointments of the system to find those associated with it--such behavior would not scale well. Retrieving associated appointments should be quick.

3.8.3.4 Design Constraints

This class was designed to make scheduling appointments easier. The Shift class presents an organized, logical way of thinking about when Medical Professionals are in the office and able to be scheduled.

3.9 Patient

This class exists as a role for a user to have. This class keeps track of the conversations and appointments held by a patient. Both conversations and Appointments register themselves with the patient.

3.9.1 Processing narrative (PSPEC)

This class is created whenever a new user that is not a Medical Professional is registered with the system. This class serves to link a user with their appointments and the conversations they have had with their Medical Professional. When a new care plan is created for the associated user, this object is notified. An instance of this class must be linked with a user before a User instance can be registered for an appointment.

3.9.2 Interface

- `getAppointments(): List<Appointment>`
- `addAppointment(Appointment)`
- `getConversations(): List<Conversation>`
- `registerConversation(Conversation)`

3.9.3 Processing Detail

The Patient class is a fairly important component of the system and supports the following functionality:

1. **Add/interact with routine appointments:** Through the UI, a patient is be able to schedule a new appointment. The Patient class generates a new instance of appointment, sets the appointment type and date/time (retrieved from the patient via UI prompts), and then associates the appointment with a Medical Professional and a Room. Upon calling `addAppointment(Appointment)`, the appointment is added to the database, the Room instance is considered reserved for the corresponding block of time, and the Medical Professional's corresponding shift is made aware of the new appointment. In order to retrieve a User's list of existing appointments, `getAppointments()` is called on the Patient instance. This particular function is called when a patient would like to view his/her appointments or when a Medical Professional/Administrative Assistant would like to view another User's appointments.
2. **Add/interact with conversations:** When a Medical Professional initiates a conversation (to speak about a care plan) with a patient, `registerConversation(Conversation)` is called on the corresponding Patient instance. This function makes the Patient instance aware of the new conversation. In order to interact with these Conversation instances (if a User wants to view/respond in a conversation thread), `getConversations()` is called on the Patient instance. Conversation instances can be selected from the returned list and then can be used for displaying conversation-related data to the User or posting new text messages (if allowed by the Medical Professional who created it).

3.9.3.1 Design Class Hierarchy

Patient conforms to the Role interface and has a list of Appointment objects and a list of Conversation objects. If a User is a patient, the User class has a reference to one Patient instance.

3.9.3.2 Restrictions/Limitations

Each patient of a practice should be associated with a single Patient instance in the system.

3.9.3.3 Performance Issues

N/A

3.9.3.4 Design Constraints

A Patient **does not** inherit from User. Rather, a User references a single Patient instance (if that User is a patient of the practice). While it may seem more intuitive for the Patient class to inherit from User, this configuration allows a User to be both a patient and a staff member at the practice.

3.10 Conversation

This class is the controlling entity for conversations between patients and professionals. These represent a set of messages that forms one interaction between patients and professionals. This collection of messages can always be commented on by the doctor, however can only be replied to by the patient if it is marked repliable.

3.10.1 Processing narrative (PSPEC)

A conversation is created and maintained by the Medical Professional class. The conversation then notifies the Patient that it is attached to of the new conversation. If the medical professional deems the conversation closed, the patient may not add a new message. Otherwise the conversation can continue to grow, accumulating more messages and files. Note that the Doctor may always add more messages to a conversation, it is only ever closed to the patient.

3.10.2 Interface

- `setPatient()`
- `setMedicalProfs()`
- `setRepliable()`
- `getRepliable()`
- `notifyOfMessage()`
- `addMessage()`

3.10.3 Processing Detail

An instance of the Conversation class is shared between a Medical Professional object and a Patient object. There are several functionalities this class supports, and some of those are only available when called by a Medical Professional:

1. **Set Patient & Medical Professional:** Upon creating a new conversation, a Patient instance must be referenced via `setPatient()`, so that the patient can begin receiving messages. In addition, the Conversation object must retain a reference to relevant Medical Professionals (for notifying about new messages), and this is done via `setMedicalProfs()`.
2. **Determine if conversation is open:** Using `setRepliable()`, the **Medical Professional** object associated with the conversation determines whether a conversation can be replied to by the corresponding patient. `getRepliable()` is called before a **Patient** instance adds a message to a conversation.
3. **Notify receiving user (Role):** When a message is successfully added to a Conversation instance, `notifyOfMessage()` is called on either the Patient or Medical Professional, so that the receiving entity can display new messages to the user.
4. **Adding a new message:** Both Patient and Medical Professional instances can add text messages to a conversation via `addMessage()` (see function 2 above for when this is false). Medical Professionals can also add files to the Conversation object via this

function. This function is triggered when a user indicates via the UI that he/she would like to add a message.

3.10.3.1 Design Class Hierarchy

Conversation does not inherit from any other classes.

3.10.3.2 Restrictions/Limitations

N/A

3.10.3.3 Performance Issues

N/A

3.10.3.4 Design Constraints

Many functions in this class rely on determining if the Role contributing to the conversation is a Patient or a Medical Professional. This part of the system must will be designed and tested carefully, to ensure that Patient instances cannot alter the conversation when they should not.

3.11 StaticResourceManager

This class acts as a manager for modifying the practice's resources. This means that when administrative assistants and medical professionals add rooms, room types, and appointment types to the system's office settings, this class is referenced. StaticResourceManager exists solely to make modifying the practice's setup more modular.

3.11.1 Processing narrative (PSPEC)

Because this class is static, no instances of it are generated by a user session. Rather, methods on this class are referenced to modify the system's representation of rooms/room types/appointment types.

3.11.2 Interface

- addAppointmentType(string) : void
- addRoomType(string, List<AppointmentType>) : void
- addRoom(string, RoomType) : void

3.11.3 Processing Detail

This class supports the following functionality:

1. **Manage system resources:** An instance of Employee or Medical Professional calls the addAppointmentType(), addRoomType(), and addRoom(), when a physical user indicates (via UI interactions) that they would like to accomplish one of these three tasks. Once added to the system, these resources can be used in generating new Appointment instances.

3.11.3.1 Design Class Hierarchy

This class does not inherit from any other class. It simply maintains a reference to Room, RoomType, and AppointmentType instances.

3.11.3.2 Restrictions/Limitations

The system should take extra care to prevent duplicate Room instances (multiple representations of the same physical room would cause scheduling conflicts) by prompting for a unique room identifier. Requesting a unique room identifier helps mitigate the potential for this issue. Additionally, RoomType and AppointmentType names should be unique, to prevent conflicts.

3.11.3.3 Performance Issues

N/A

3.11.3.4 Design Constraints

This class exists solely to provide a more structured way of interacting with/modifying the 'setup' of the office. In order to add Room, RoomType, and AppointmentType instances, calls are made to this StaticResourceManager.

3.12 AppointmentManager

This class is responsible for maintaining all instances of Appointment. AppointmentManager abstracts other components away from directly speaking with the backend when retrieving/filtering Appointment instances.

3.12.1 Processing narrative (PSPEC)

When a new user sessions occurs, a single instance of AppointmentManager exists for retrieving Appointment instances.

3.12.2 Interface

- `getAllAppointments(): List<Appointment>`

3.12.3 Processing Detail

This class supports the following functionality:

1. **Retrieve a list of Appointment instances:** `getAllAppointments()` is called when other components of the system need to see what Appointment instances already exist in the database (functionality related to generating new and showing existing Appointments will eventually require this class).

3.12.3.1 Design Class Hierarchy

This class does not inherit from any other classes but follows the singleton (manager) paradigm.

3.12.3.2 Restrictions/Limitations

N/A

3.12.3.3 Performance Issues

N/A

3.12.3.4 Design Constraints

While this class does not actually define a new data structure, it exists to provide a layer of abstraction away from the database level.

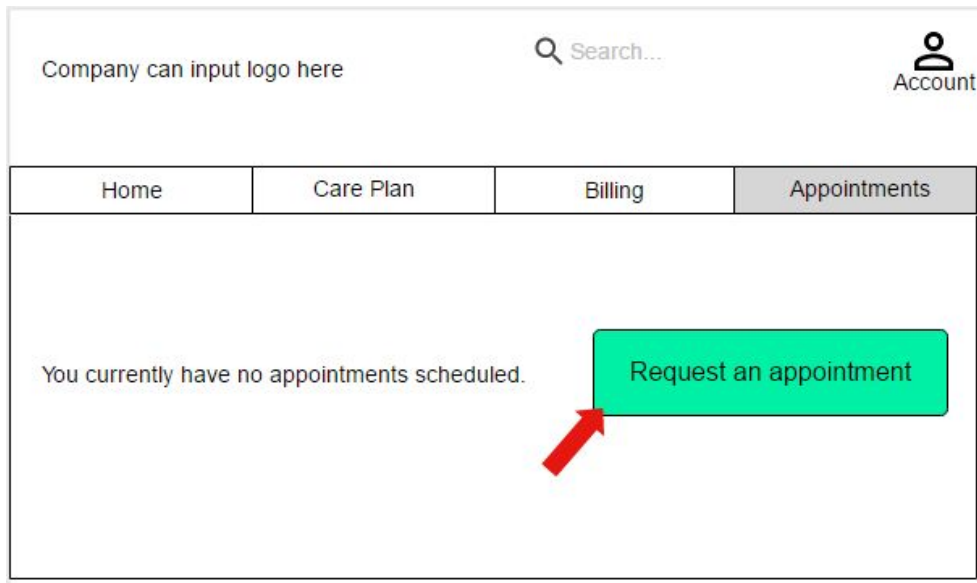
4.0 User interface design

When it comes to PatientNow!'s user interface we've prioritized functionality and ease-of-use. We understand that many of the patients who will be using our software were raised in a time before the internet and so we've tried to make usage of PatientNow! as intuitive as possible.

4.1 Description of the user interface

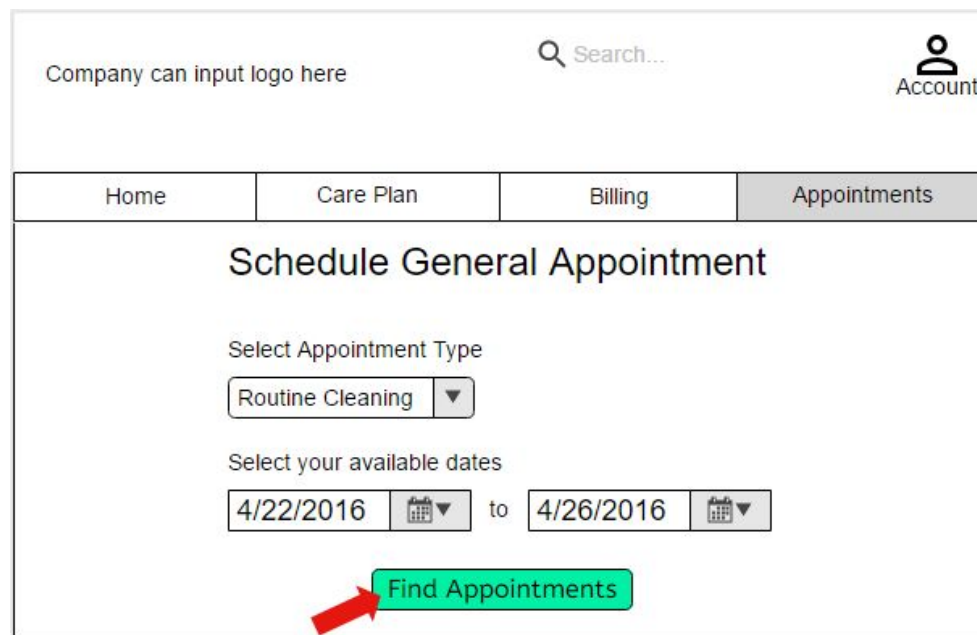
Our UI has two different views depending on the user. A patient's view has four major tabs: Home, Care Plan, Billing, and Appointments. Once a patient is logged in, these tabs along with the practice's logo, a search bar, and an account button, are always present at the top of the screen! Medical professionals will see a 'Schedule' and 'Patients' in place of the Care Plan and Appointments tabs. They will also see a 'settings' button which provides them with options for staff, room, and appointment management. Important buttons are highlighted by a calming seafoam green made all the more apparent thanks to our clean white design.

4.1.1 Example Case 1: Scheduling a general appointment



This screenshot shows the top navigation bar with a search field and an account icon. Below it is a tab bar with 'Home', 'Care Plan', 'Billing', and 'Appointments'. The 'Appointments' tab is selected. The main content area displays the message 'You currently have no appointments scheduled.' and a green button labeled 'Request an appointment'. A red arrow points to this button.

Step 1: Under the Appointments tab, patient clicks 'Request an appointment'



This screenshot shows the 'Schedule General Appointment' form. It includes a title 'Schedule General Appointment', a 'Select Appointment Type' dropdown menu with 'Routine Cleaning' selected, and a 'Select your available dates' section with date pickers for '4/22/2016' and '4/26/2016'. A green button labeled 'Find Appointments' is at the bottom, with a red arrow pointing to it.

Step 2: Patient chooses appointment type and available dates, then clicks 'Find Appointments'

Company can input logo here

Search...

Account

Home

Care Plan

Billing

Appointments

Select Appointment

| | | |
|-----------|-------------------|--------|
| 4/22/2016 | 9:30am - 10:00am | Select |
| | 11:30am - 12:00pm | Select |
| | 2:30pm - 3:00pm | Select |
| | 4:30pm - 5:00pm | Select |
| 4/23/2016 | 2:30pm - 3:00pm | Select |

Step 3: Patient scrolls through list of available times and clicks 'Select' on the one they would like.

Company can input logo here

Search...

Account

Home

Care Plan

Billing

Appointments

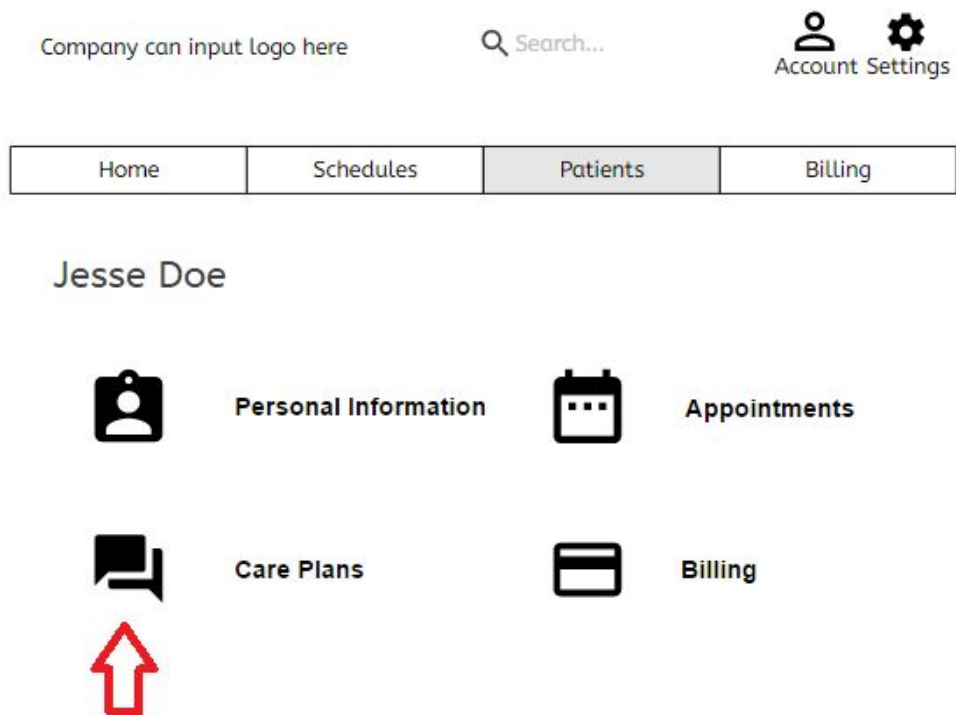
Below are your scheduled appointments.

Request an appointment

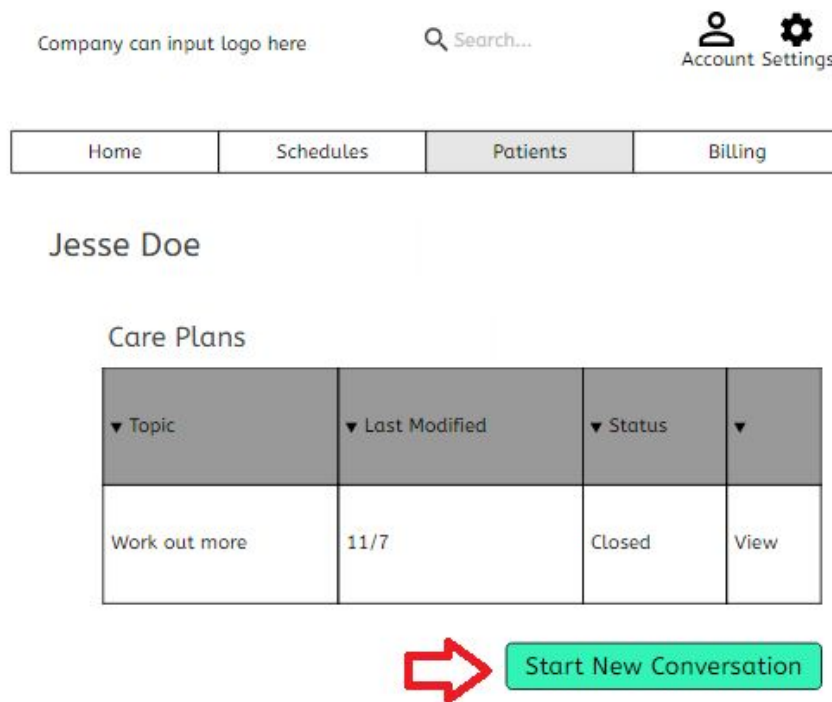
| ▼ Type of appointment | ▼ Date | ▼ Time |
|-----------------------|---------|-----------------|
| Routine Cleaning | 4/22/16 | 2:30pm - 3:00pm |

Step 4: Patient is redirected back to the Appointments tab where they can now see their upcoming appointment

4.1.2 Example Case 2: Sending out a new patient care plan



Step 1: When viewing the profile of a patient, the medical professional or administrator can view the conversations about that patient's care plan by clicking on the "Care Plans" icon.



Step 2: Now the medical professional or administrator is viewing the patient's care plan conversations. User then clicks button that says "Start New Conversation."

Company can input logo here

Search...

Account Settings

Home

Schedules

Patients

Billing

Jesse Doe

New Care Plan

Enter title.

Provide a short description.

Start the conversation!

Can the patient respond?

Yes

Start conversation

Step 3: A form appears for a new conversation between the medical professional and patient. User fills relevant information into blank fields in the form, and they can choose whether or not the patient will be able to respond to the initial post. The user then presses "Start conversation."

Jesse Doe

Eat more oreos last updated 11/11/2016

Close conversation

Hello Jesse,

It seems you are too stressed, and need more dessert in your life. Since you are vegan, an inexpensive option is Oreos! I'd recommend one with almond milk per day.

Enjoy,
Paula

Add new comment.

Submit

Step 4: The conversation is posted with an option to add a new comment. The user can also close the conversation.

Jesse Doe

Eat more oreos last updated 11/11/2016

Open conversation

Hello Jesse,

It seems you are too stressed, and need more dessert in your life. Since you are vegan, an inexpensive option is Oreos! I'd recommend one with almond milk per day.

Enjoy,
Paula

This conversation has been closed. Click [here to open](#).

Step 5: This is the view if the user closes the conversation. They have the option to open as well.

4.1.3 Other Example Screens

Registration

Insert Logo Here

Register today for easy access to your care plan and medical history.

| | |
|--|---|
| First Name | Last Name |
| Email Address | Medical ID |
| Password must be at least 8 characters long and contain one of each of the following: <ul style="list-style-type: none">- capital letter- lowercase letter- number- symbol (!@#\$%^&*?+=) | <div>Password</div> <div>Confirm Password</div> |

Register

Log in

Insert Logo Here

Welcome!

Email Address

Password

Login

Forgot Password?

Account Settings

Company can input logo here

Search...

Account

| | | | |
|------|-----------|---------|--------------|
| Home | Care Plan | Billing | Appointments |
|------|-----------|---------|--------------|

Personal Information

Name: Tim Kearns

Email: tkearns@calpoly.edu

Phone Number: 916-832-7015

Preferred contact type: email

Password: *****

[edit](#)

[edit](#)

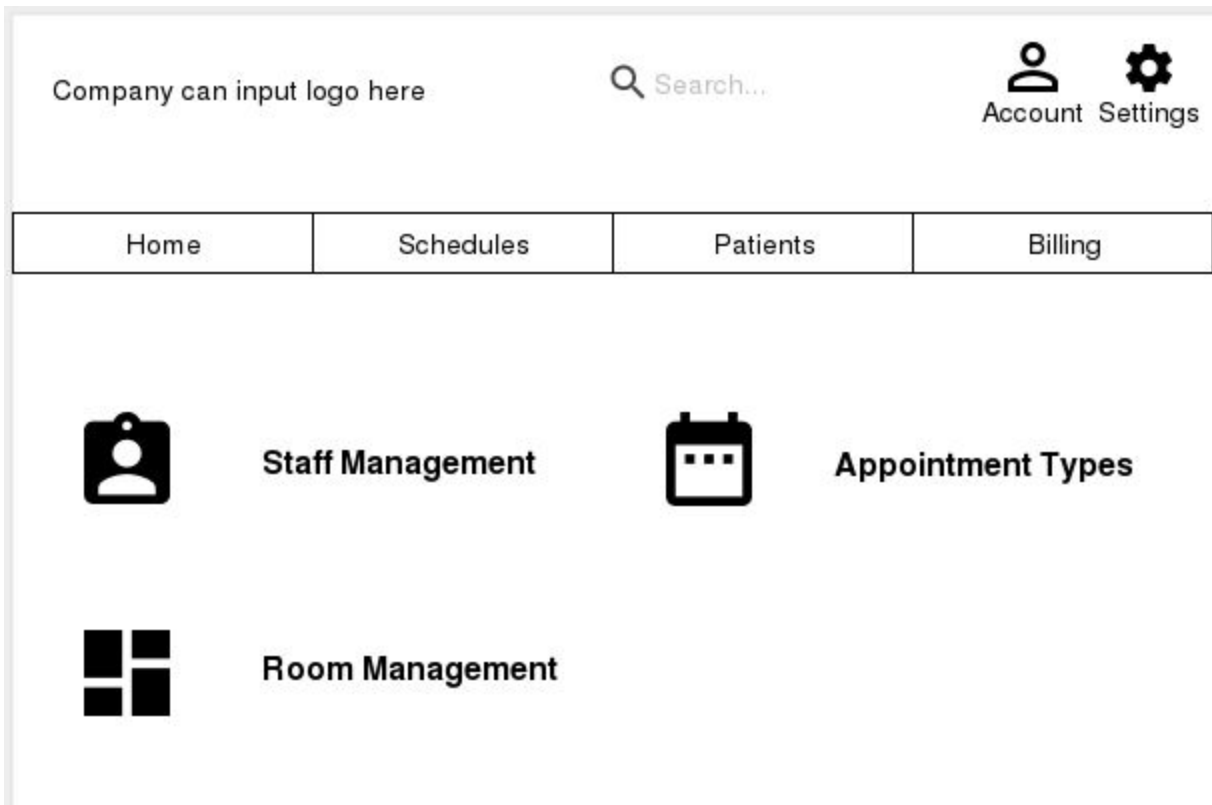
[edit](#)

[edit](#)

[edit](#)

Billing

Medical Professional Settings



4.2 Interface design rules

1. All colors must come from our list of approved PatientNow! colors.
2. Colors are used to highlight important areas, not just aesthetically
3. Consistency is key. For example, buttons like 'Account Settings' must appear on every screen in the same place.
4. Utilize space properly. For example, if a page only has 2 buttons they should be big buttons.
5. Keep it simple whenever possible.
6. Use an icon instead of text when a button's function can be cleanly represented as such

4.3 Components available

- Buttons
 - Buttons will allow the user to interact by clicking on it. It will trigger an event or direct the user to a different page.
 - This component will facilitate the majority of user interaction on our website.
- Calendars
 - Calendars are used to display doctor availability and appointments in an easy-to-understand format.
- Checkboxes
 - Checkboxes allow users to select one or more items/options on which to make an action
- Drop-down Fields
 - Drop-down fields are used to allow the user to choose only one of multiple options.
- Icons
 - Icons will be used throughout our website to help users distinguish sections or buttons of our website, improve navigability, and employ a cohesive style.
- Input Fields
 - Input fields are used throughout the website to collect user information that will be saved in the system for future use.
- Scroll View
 - Scroll views can be used when the content to be displayed is dynamic and might be longer than the container that contains it.
 - It improves user experience by limiting the content displayed to what the user wants to see.
- Tables
 - Tables are used to display user-friendly representations of our databases, such as our collection of patients and scheduling information.
- Text Fields
 - Text fields provide text content for our users

5.0 Appendices

Presents information that supplements the design specification.

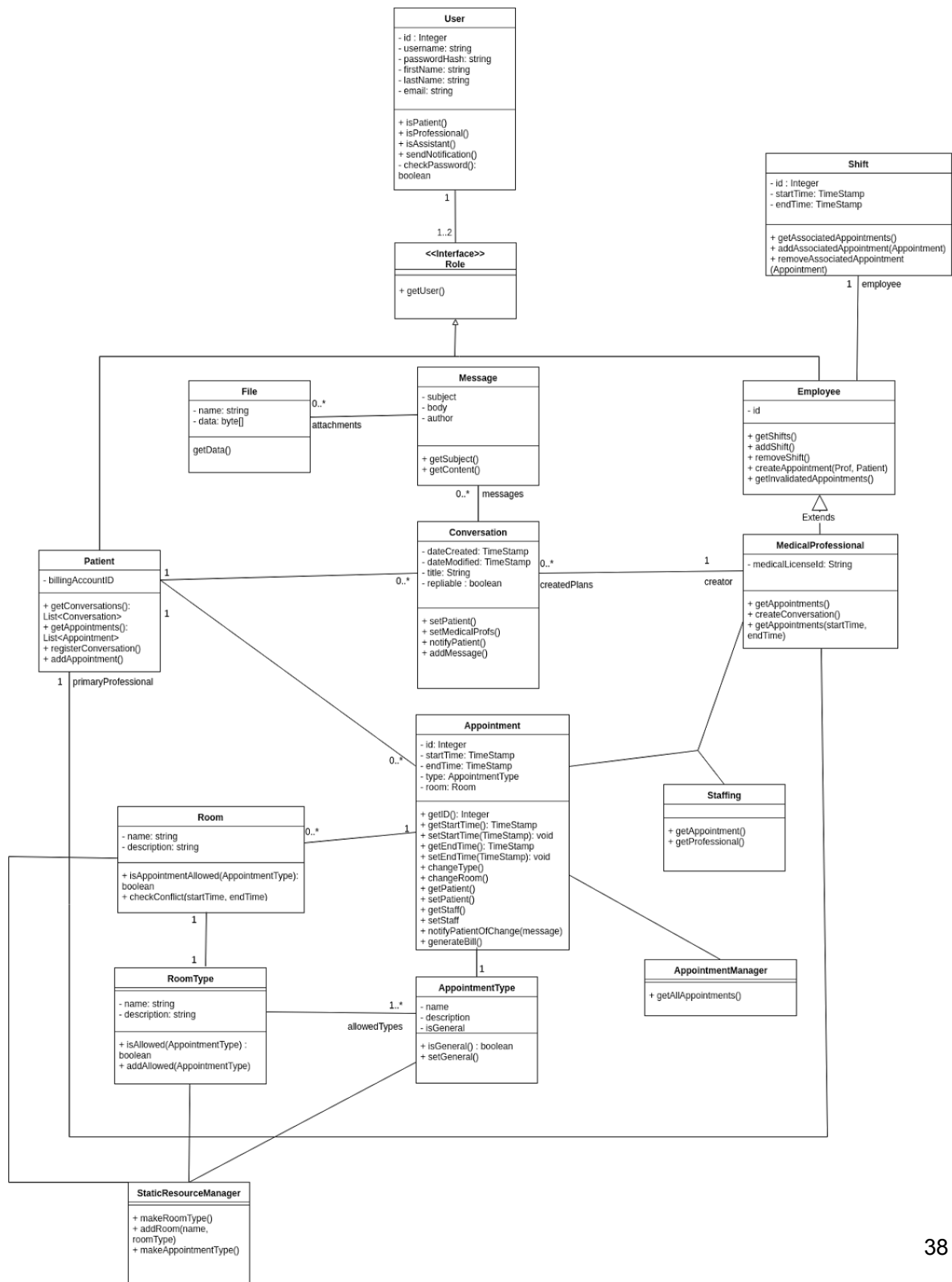
5.1 Packaging and installation issues

The system must be installed on a system hosted by the practice that wishes to use it.

This installation will take a member of the team having temporary access to the system.

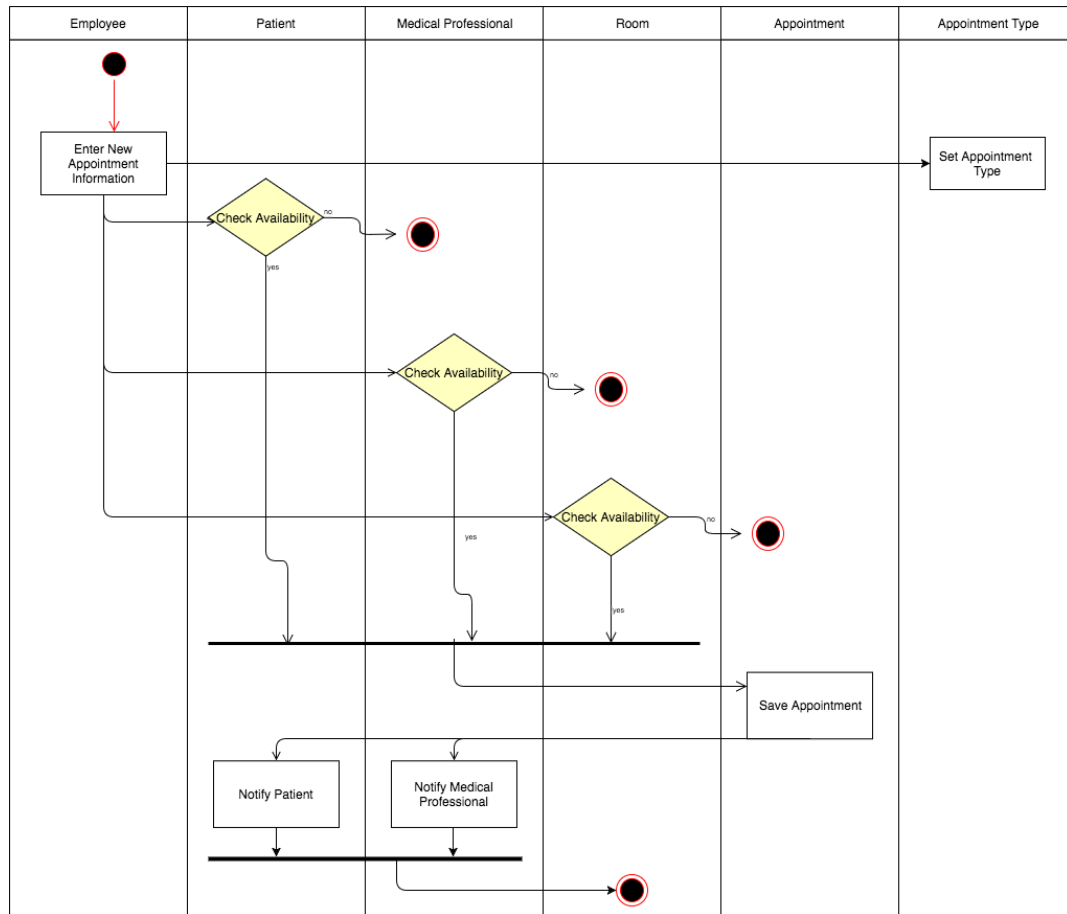
5.2 Diagrams

5.2.1 Class Diagram

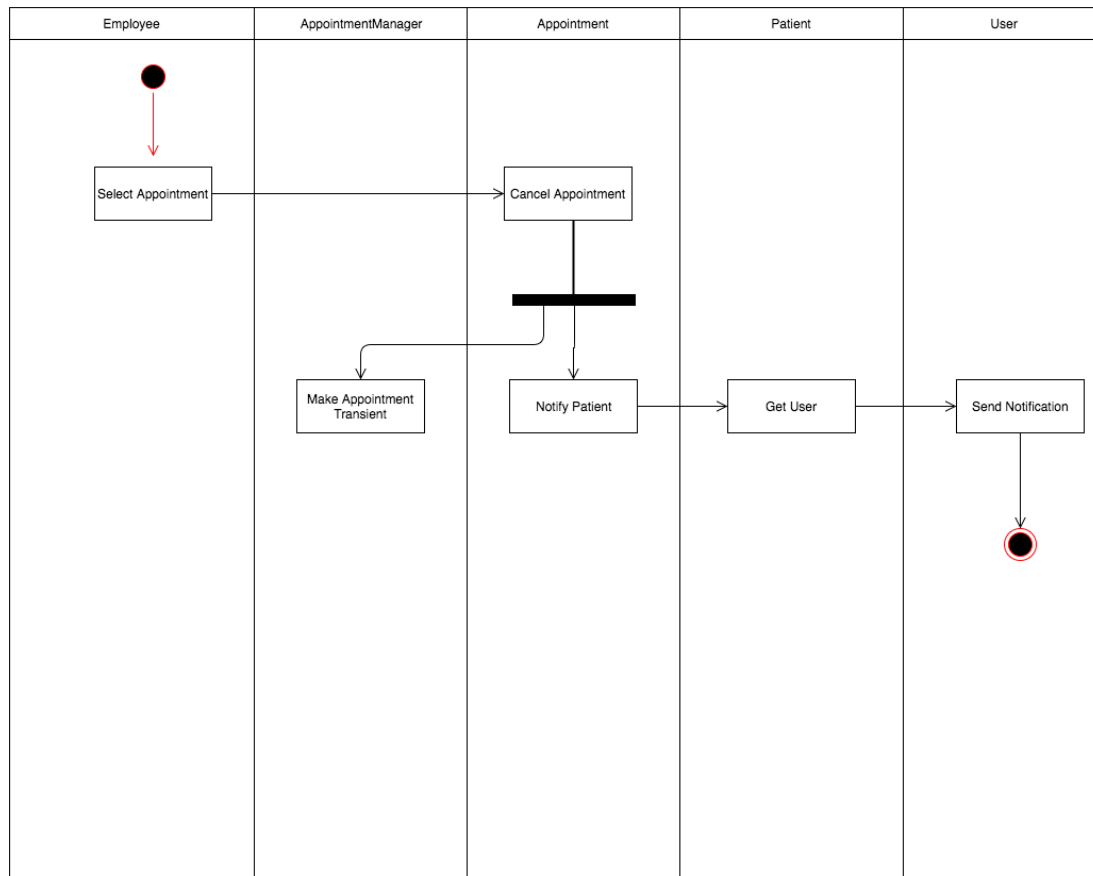


5.2.2 Activity Diagrams

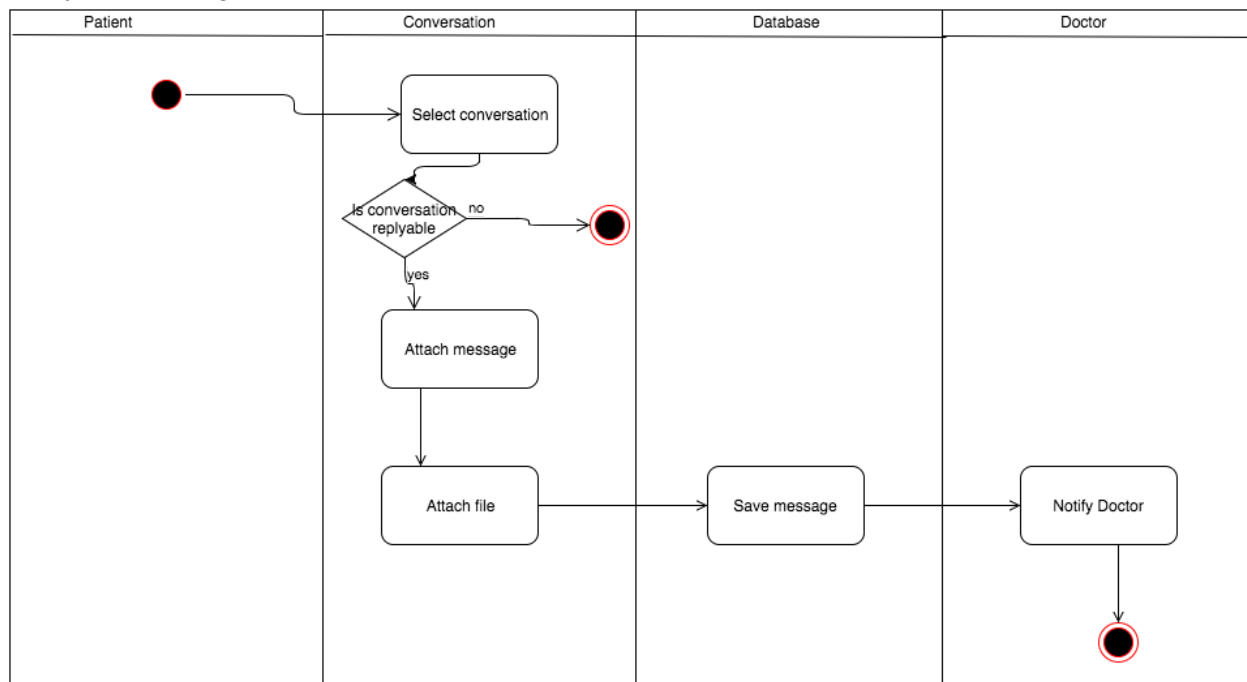
Add Appointment (Employee) Activity Diagram



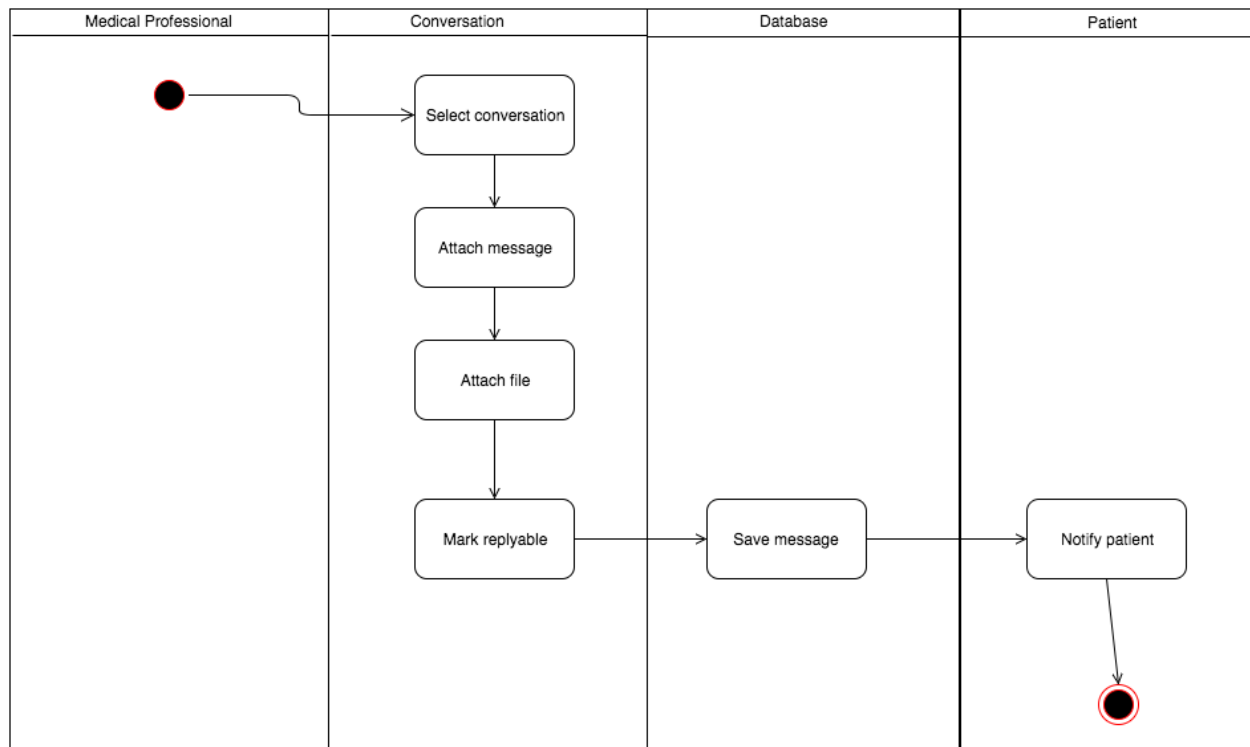
Remove Appointment



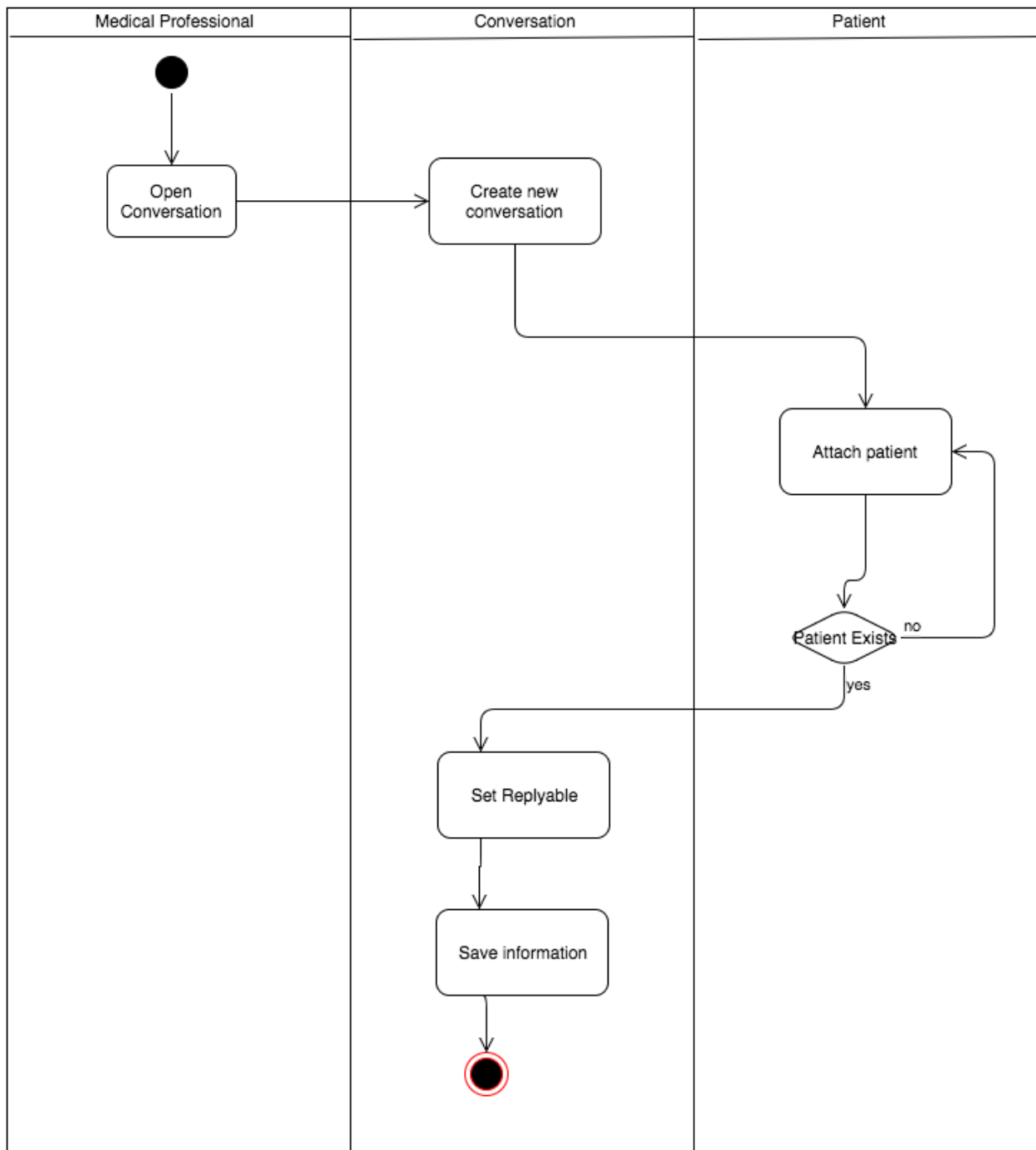
Reply To Message



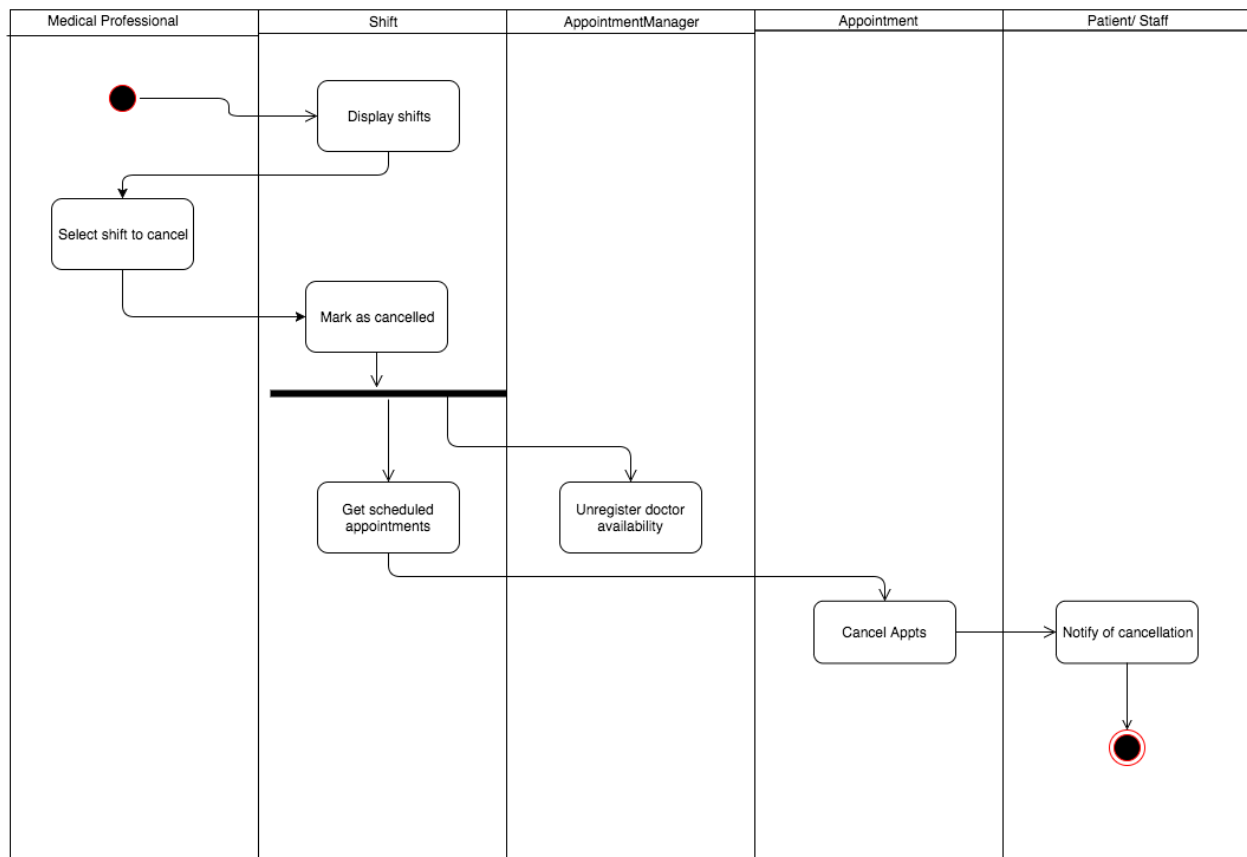
Medical Professional Sends Message



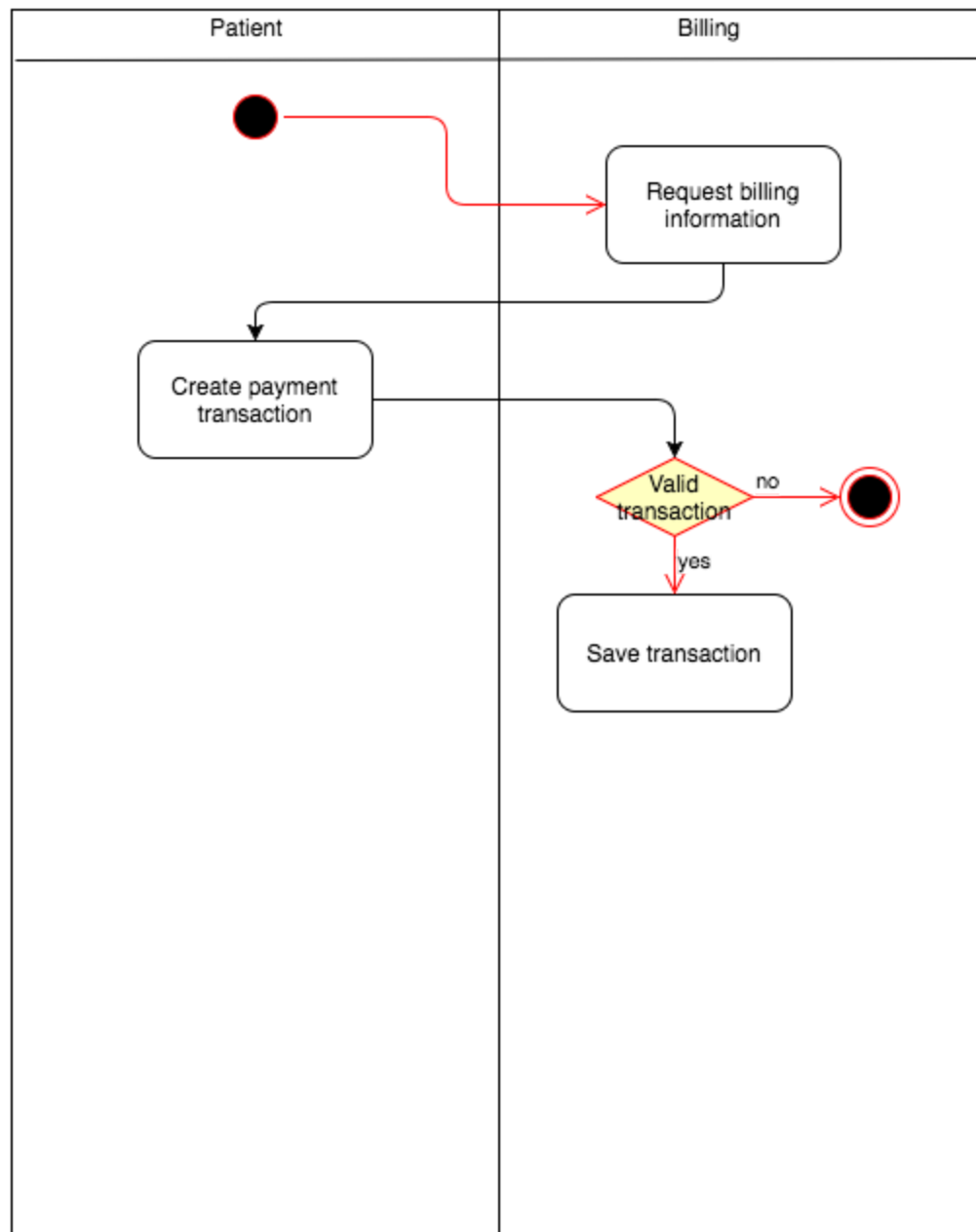
Create Conversation



Cancel Shift



Pay Bill



Add Appointment Type

