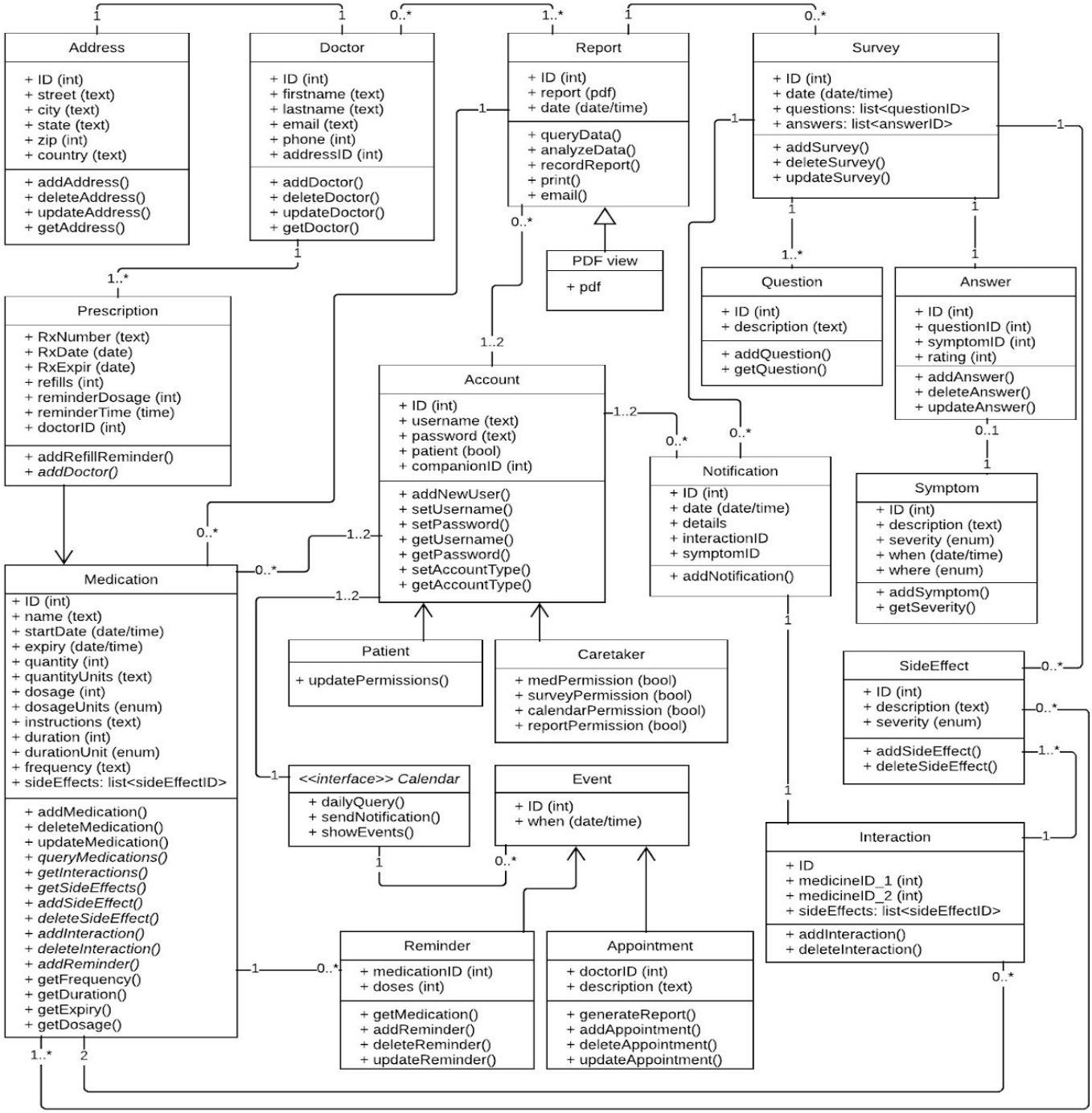
Homework 4

Mohammed Algadhib
algadhim@oregonstate.edu
Riley Kraft
kraftme@oregonstate.edu
Diego Saer
saerd@oregonstate.edu
Johnny Chan
chanjoh@oregonstate.edu

1. OBJECT-ORIENTED UML CLASS DIAGRAM

The following diagram is a depiction of the attributes, methods, and relationships of the various classes within our software system. Edits were made from our previous version of the UML to better modularize our plan for implementation, as well as further touch on inheritance. Lastly, additional methods and attributes have been added in order to fulfill some need of our architecture that were discovered in HW3.



2. IMPLEMENTATION PACKAGING

For the logical packaging of the UML class diagram above, the implementation of this system can be divided into 6 manageable parts with the least amount of coupling and, subsequently, the greatest amount of cohesion possible.

First and foremost, the establishment of a user account must take place before any other parts of this system can even become relevant. Thus, the first package includes the following classes: Account, Patient, and Caretaker. Patient and Caretaker inherit from Account, thus these must be packaged together. The purpose of this package is to create, authenticate and manage user accounts. Attributes and methods within this package are not dependent on attributes or methods from other packages. However, there is an instance where the Report class is dependent on the Account email attribute for sending a report via email. In this instance, there is an occasion for data coupling, one of the lowest levels of coupling. Otherwise, this package reflects a high level of cohesion in which a connected Patient and Caretaker manage the same data for a shared purpose.

The second, and largest, package provides all the data for which the other packages utilize in their methods to create their own resulting data. This package consists of the Medication, Prescription, Doctor, Address, Interaction, and SideEffect classes. The Prescription class inherits from the Medication class, thus these must be in the same package. Similarly, a Doctor object can only be created through the Prescription class, i.e. a user can only add a doctor for which they have a prescription. Similarly again, an Address object can only be created through the Doctor class. Currently, there is no need to store Address objects for a Patient or Caretaker object, however the Address of a Doctor is required if the user wishes to include this information in their doctor's appointment reminders. Lastly, the Interaction and SideEffect classes must be part of this package because their data are derived from the Medication class methods and name attribute which query the Medications DB API. The methods and data within the Interaction and SideEffect classes are solely dependent on the Medication class. Furthermore, Interaction objects provide additional data to the SideEffect class records as interactions between medications may have their own set of known side effects. Therefore, this package, internally, exhibits mid-level coupling as the various classes require data and/or methods from one another, but they do not write/edit the same data. Similarly, this package, internally, exhibits mid-level cohesiveness as the various classes have procedural-to-temporal cohesion. These classes don't necessarily use the same data, however they do rely on each other to execute methods which create each other's data. In regards to inter-package coupling and cohesion, this package again has some mid-level coupling and, subsequently, mid-level cohesion. Particularly, in regards to the addition of Reminder objects. Data derived within this package, through the Medication interface, is used to create calendar Reminder objects in the next package discussed.

The third package consists of the following: the Calendar interface and the Event, Reminder and Appointment classes. The Reminder and Appointment classes inherit from the Event class, thus these must be packaged together. This package features common coupling and, subsequently, communicational cohesion as the Reminder class reads attribute records derived in the Medication package. However, the Reminder class does not edit or write this same data. Similarly, the Appointment class reads attribute records derived in the Doctor and Address classes. Again, however, the Appointment class does not edit or write this same data. Lastly, although this package does include a daily query method which could trigger an alert, though it is not the same as a Notification object. This alert is only outputted to the client interface and is not created as a Notification record, thus there is no dependency on the Notification class to carry out this method.

The fourth package consists of the Survey, Question, Answer, and Symptom classes. The Survey class is entirely dependent on the Question class, since without questions there cannot be a survey. Similarly, the creation of Answer objects is entirely dependent on the Survey class. Without the completion of a survey, there are not answers to record. The inter-packet coupling for this package is found in the reliance of the survey questions on the records created from the SideEffect class. The possible symptoms presented to the user when taking a survey are derived from the SideEffect class records. Hence, this package exhibits stamp coupling as the Survey class is provided structured data from the SideEffect class records. When recorded, a symptom references the ID of a known side effect stored in the system database. However, the recording of symptoms does not solely rely on the SideEffect records, since the user may add new Symptom objects of their own volition. Hence, this package features low coupling and, subsequently, high cohesion.

The fifth package consists of one class, Notification. The Notification package is utilized as an alert system only by the Account package. However, the Notification package is dependent on attributes and methods within the Interaction and Survey classes. While this dependency does require a mid-level amount of coupling, control and/or stamp, these classes do not go as far to read/write the same data. The addition of an Interaction calls the Notification class to add a notification with structured data sent from the Interaction class. Similarly, the Survey class analyzes records of the Answer class to structure data and call the Notification class to add a new notification. As a result, there is also mid-level cohesion between these classes. The Interaction and/or Survey classes execute their methods of creating and/or analyzing their attribute records, then call on the Notification class to execute a method and create a record from structured data sent from the Interaction/Survey class.

The sixth and final package also consists of only one class, Report. The Report package queries data provided by all other packages in the system. However, the report does not alter any of the data from any of these packages. Rather, the Report package draws unstructured data from the other packages and analyzes it to create a Report object, thus exhibiting low-level coupling. As well, the generation of a report calls the various other packages to get() their respective data records. The majority of the data records created from the other packages are derived to provide the data required to analyze a Patient's level of health on their current medications. Since the main purpose of this system is to monitor and report the well being of an elderly patient on various medications, it can be discerned that this package has the highest level of cohesion within the UML class diagram.

3. INCREMENTAL VS ITERATIVE

The goal of the system is to assist in providing quality healthcare. This is done by providing two main functions:

1) Keeping track of the patient and their medication (carried out primarily by the mediation and survey classes).

2) Notifying the patient, caretaker, and/or healthcare provider on the patient's status, important events, and/or any potential problems (carried out primarily by the calendar, report, and notification classes).

Keeping track of the patients without informing the involved parties of anything important is pointless. It is also impossible to inform the involved parties without keeping track of the patient and medications. The point is that the value of the system is spread out throughout all subsections and this supports a preference for an iterative development process.

There were other considerations that were looked at and they are listed below.

Reusability: Code reusability is not a key quality attribute for the system so the potential to reuse code was not given great consideration in determining the development method.

Security: The need to keep client data safe in both network communications and physical storage is of paramount importance. Because if this it would seem to imply that every component of the system should be as secure as possible before release and therefore development be iterative.

Interoperability: This key quality attribute stresses that the system should communicate with external components well. In this case iterative development is more useful as external components may be replaced with new versions, e.g. new database system or a better API, and the client system has many subsystems that would all need to be updated to work with these.

Reliability: This is the final key quality attribute of our system (usability is not being considered since it depends on the GUI) and it would require our system to work as expected close to 100% of the time. It may even imply adding redundancies and extra exception handlers in the code to make sure everything runs as smoothly as possible. This would seem to imply a preference for incremental development as each sub system must be as robust as possible when released to the public.

We are left with a situation where two of the three most important quality attributes seem to show a preference for incremental development. Unfortunately, as stated above, having part of the system by itself is pointless. Therefore, extra precautions must be taken to ensure the system is as secure and reliable as possible upon release of every iteration of the software.

4. DESIGN PATTERNS

Our software is based on the implementation packaging that we could divide the system into sections and each section is using a method of design pattern. These are general design patterns for each package, however, a package might have more that one design pattern. More design patterns were considered, the following made the "final-round" for consideration. Some of these design patterns, for reasons explained below, did not make the cut for our software system.

4.1 FACADE METHOD

The facade design pattern is a method of hiding the complexity of the system under a simple interface. The system consists of internal complexities regarding design and implementation within the client that the user will not see. The user accesses the databases and servers through graphical user interfaces, and completes tasks without having to know the internal workings of the system. Users cannot access or modify what's under the hood of the software. Users can see the icons to add a medication, take a survey, view the calendar, view their notifications, generate a report, etc. with only a few buttons. However, they cannot see the large body of code that's used to create those buttons or their respective functions. Users are also taken to different pages when events are triggered. For example, a user can access the calendar which will take them to the calendar page from the homepage of the interface. As well, users receive notifications for upcoming appointments, dosage reminders, and medication reaction warnings, though they do not have to be concerned with the complexities behind the design of the interface.

4.2 OBSERVER METHOD

The observer method is a design patterns where there is an object that keeps track of its dependant and is invoked

when there are any changes in their states. In simpler words, the observer method is an implementation of an event handling class. One of the strongest features in our software is having a feature that functions as an observer. The observer will track for changes in the state of the client and the server. When the observer has found changes, it will trigger events that resulted from the changes. For example, the observer for our software will send notifications to the user when an upcoming appointment is scheduled. This event is triggered when the appointment is within a 24-48 hours and an alert is initiated. This is due to the change in state of events of the patient user. Another example is when reports are being generated and submitted for doctors. Users will also receive notifications about report generation and submission to the doctor. These observers will look for patterns and changes at all times. When one of the events are triggered, it will create related events as notifications to the users. Moreover, one of the main and most important functionality that this system will do is to notify the user whenever there is a new interaction added to the list. The perfect design pattern to detect this change is the observer method.

4.3 INTERPRETER METHOD

The interpreter method is a design pattern that serves as an interpreter to transfer data representation to be processed by different systems, or even between parts of the system. Our software has the capability to read instructions written in another language or outputted in a different file format. Our databases can take in requests by the GET method from the client and interpret them as queries using SQL parsing in the server. The server can store them if the user makes similar requests. Medications that were added on to the user's account are interpreted by the database and outputted to the client. The client can get these files back and see them outputted in a format that our software is programmed to present. Surveys that were submitted as expressions to the database can also be interpreted as queries with SQL. The symptoms that resulted from the answers to the survey is interpreted by the local database and the client. Moreover, since this system will be requesting data from different APIs and then store the data in the system's database, the interpreter will receive the data from the APIs as JSON objects, parse them, and then transfer them as SQL commands to store them in the database using GET/POST requests. As a result, the interpreter design pattern is one of the important patterns that will be used in implementing this system as it handles a significant part of the system functionality.

4.4 COMPOSITE METHOD

In the composite design method, objects can be treated the same as a single instance of that object, and they can be thought of as a tree structure. The composite holds the child components and runs related child operations. For example, a Medication is the composite which consists of Prescription objects as its child components. It can modify the child class, variables, and methods as it needs. An applicable example from this system about the composite design pattern is the patient and caretaker classes, these two objects are similar in functionality, but differ in permissions.

4.5 TEMPLATE METHOD

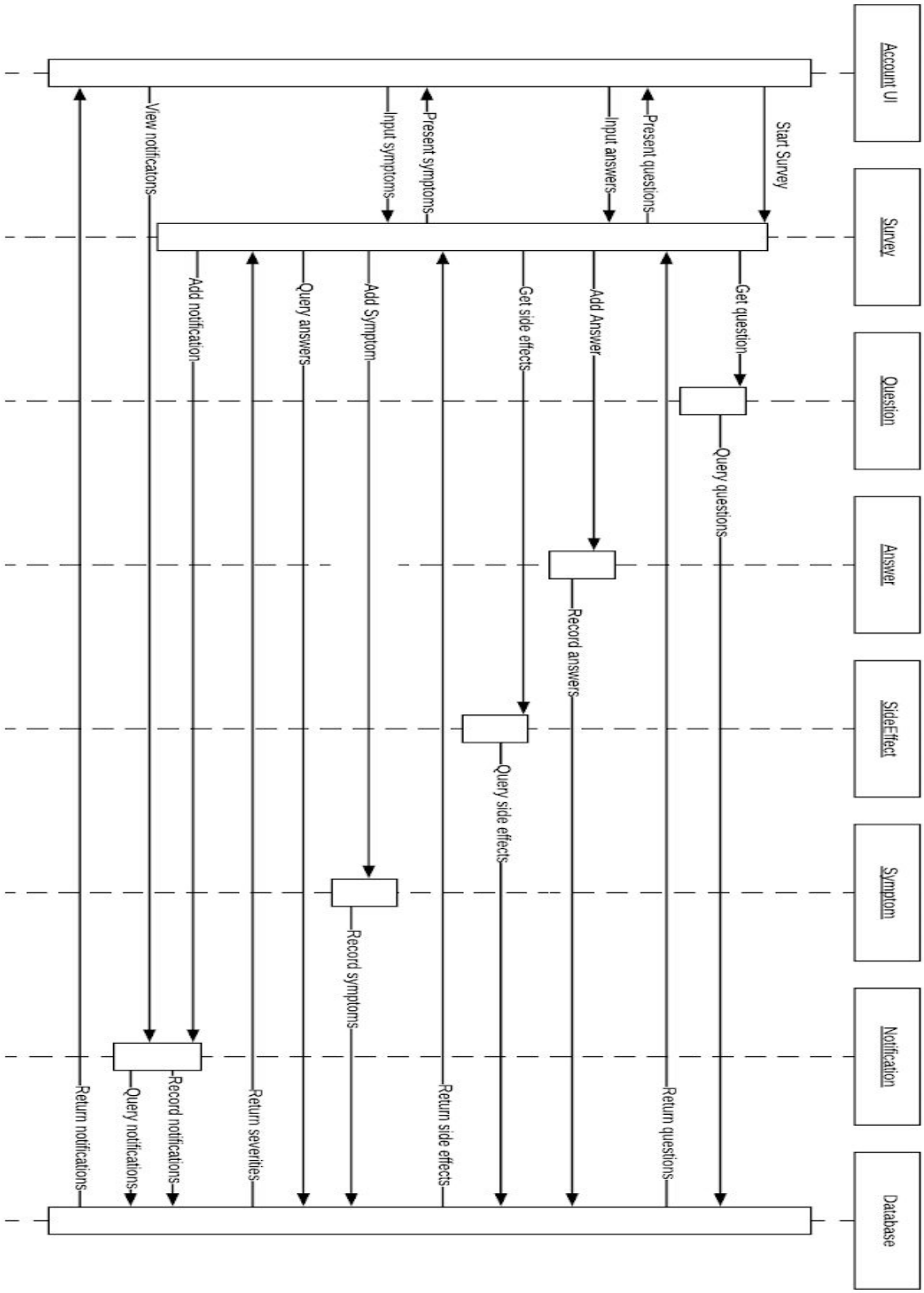
In the Template design pattern there is a main object that implements base functionality. Then, there are subclasses implemented under that main object that can utilize the methods from the main object. However, these subclasses might require alterations or additional algorithms to the base class methods in their own implementations. Initially, we believed this design pattern might apply to several of our classes that require inheritance or abstract methods. However, upon further inspection, we realized that we did not require alterations to main class functions. Instead, we merely needed to access them from inherited or abstract subclasses. In this system design there are no cases that require this design pattern in their implementation.

4.6 MEMENTO METHOD

The memento design pattern is a method of implementation that gives the ability to record the previous state of the application and go back to them if necessary. In other words, the memento design pattern provides the ability to undo changes while using the application. Based on the design of this system, recording the application state while inputting data, like taking the survey, is not necessary since the data wont be stored unless the user submits the survey after answering all the questions. As a result, we would not in fact have to restore previous versions of the application or data, hence the memento design pattern is not necessary in the design of this system.

5. SEQUENCE DIAGRAM

The following diagram depicts the sequence of events for use case 2, taking the daily health survey. This diagram, however, delves more into the specifics of the the flow of data and types of methods called on between our software classes in order to achieve the use case 2 functionality.



6. INTERFACES

In the architecture of this application, communicating with external modules is essential to get medication information from trusted sources and storing the data in safe databases. More importantly, the user is also considered to be an external source of interaction to the system. As a result, to insure the maximum interoperability between different systems, the interaction points must be defined as interfaces and contracts so that communication is done successfully. The interfaces to our system could be generalized into two main categories, the first interface will be for the user interaction with the system, and the second is in where the system interfaces with service providers i.e APIs and DBs.

In the user interaction interface category, the main interface in this category is the UI, the user interface, since this application is designed to be used in multiple platforms, mainly PCs and mobile devices. As a result, there will be two graphical interfaces for the application depending on the platform used to interact with the application. Thus, some of the interfaces might slightly differ, for example, when using a touch screen on a mobile device vs. using a mouse on a PC. However, these two preconditions can be altered between both platform as they serve the same purpose. This interface serves as the interaction point between the user and the application to insert data and initiate activities. For example, through this interface the user can input medications, doctor appointments, and start the daily heath survey.

For each interface there has to be some expectations as pre- and post- conditions must be satisfied in order to complete the process successfully. In the user interface, the most important precondition is that the user is logged in with an active account with the proper permissions to insert data or launch activities. The post condition of the user interface is that the data inputted is recorded and the requested activities are launched.

Moving to the service providers interfaces, in this category the interfaces discuss the connections between the components of the system's architecture. In this case, from the system architecture we have the following modules: mobile/web app, local database, Optical Character Recognition (OCR) API, web service, medication DB API, and database server. The interfaces between these modules are the following:

Mobile/web app and Web Service: this interface is to request data from the web server to be presented to the user or send inputted data from user to the web server. The precondition of this interface is to make requests with the proper protocols, GET to get data and POST to send data. The post condition of this interface is that the mobile/web application receives the requested data or an acknowledgment that the sent data was received successfully.

Mobile/web app and Local Database: This interface provides the mobile application a local access to the database to request and store information related to the patient. The precondition of this interface is that the user has the proper condition to access the database. In addition, the data routed is under the defined tables. The postcondition of this interface is that after a successful data request the user can utilize the requested data. And after a successful data update the data will be stored to the local database and synchronized to the system's database.

Mobile app to OCR API: This interface serves as a tool to translate the text on a picture of a prescription label into text to be used to register the medicine to the medication list. The preconditions for a valid request to this API is that the request contains the authorization code for the API and an image of the prescription label. After a valid request, the API will respond with a JSON object containing the text detected on the image, which the client can parse and input into the data's respective fields in the interface.

Local Database and Web Service: This interface is used to synchronize the local database with the database server. The precondition of this interface is the validity of the data in the local database, in other words, there are no alterations or modifications to the database server contents. The post condition is the local database is synchronized with the database server.

Web server and Medication DB: The purpose of this interface is to request medication information from the medication API. For a successful request via this interface the precondition is that the web service send a request to the api server with valid medicine names and, if necessary, an authentication to be able to communicate with the API. Upon the valid request, the post condition is that the Medication DB will send the information about the requested medicine to the web service server module.

Web service and database: This interface connects the database server with the rest of the system through the Web Service module. The preconditions of this interaction is that the server is requesting or storing data from or to a table that was defined in the database with proper access permission. The post condition when requesting data is receiving them and, when storing, the data will be in the database and can be accessed in the future.

7. EXCEPTIONS

Occasionally the system will run into unusual events that must be handled in a special manner. The following is a list of these.

No network connection detected: The system will warn the user a connection to the internet could not be established and that full access to the user's data is not available. It will also warn the user that any new medications added cannot

be verified for possible interactions. Finally, it will recommend the user connect to the internet before using the system. If the user persists in using the system, any new objects created, updated, or deleted will be time stamped in the local database so that when the system can sync up with the web services, these new objects can be processed correctly.

Sync with external database or API fails: If the web services return a sync failure message to the system, then the system will try up to two more times to sync. If these additional attempts fail then then the system informs the user that there exists a problem connecting to an external server and that full access to the user's data is not available. It will also warn the user that any new medications added cannot be verified for possible interactions. If the user persists in using the system, any new objects created, updated, or deleted will be time stamped in the local database so that when the system can sync up with the web services, these new objects will be processed correctly.

Sync with external database or API times out: If after a certain amount of time new data sent to the external database does not return with a confirmation of success or failure. The following will happen:

1) The system will first check if an internet connection is established, if not then the no network connection handler will run.

2) If it does find an internet connection then it will ping the web services.

a) If an answer to the ping is not returned after a certain amount of time, then the system informs the user that there exists a problem connecting to an external server and that full access to the user's data is not available. It will also warn the user that any new medications added cannot be verified for possible interactions. If the user persists in using the system, any new objects created, updated, or deleted will be time stamped in the local database so that when the system can sync up with the web services, these new objects will be processed correctly.

b) If the ping is returned then the system will try to sync once again. If a timeout or sync fail happens, then the event is handled as if a ping had originally not been returned.

Patient restricts caretaker's access to a module while the caretaker is using it: A signal is sent to the web services that the caretaker's access has been restricted. The caretaker's device will then sync with the web services, if they are online, and then a message will appear for the caretaker that says their access to the module has been restricted. The system will then return the caretaker to the home screen and deny them future access to the module until permissions are restored.

The system is restarted after a crash while a form was being worked on: To handle this the system must keep track of whether data that is written to the hard drive has been completely written or not. Perhaps when working with any objects it can create a special file that contains information about the object. If an operation is successfully completed, then this file is deleted. If the system crashes and then upon restart of the system this special file is found, a warning to the user is presented giving the name of the operation that ended unexpectedly, e.g. adding a new medication, and taking the user to the incomplete form.

Incorrect data type is input into a form: The program will not continue with object creation, e.g. form submission or report creation, but instead will revert to the form. A message will appear to the user showing the names the fields with incorrect data. When the user goes to the field with incorrect data, a few samples of data entered in the correct format are shown beside the field.

Object creation, delete, or update fails: The system will try to create/delete/update the object two more times. If the second of these extra attempts fail, a popup will be displayed to the user that says the current action has failed and that the program should be restarted if this continues.

8. REFERENCES

[1] DrugBank API. [Online] https://docs.drugbankplus.com/v1/#introduction

[2] Free OCR API. [Online] https://ocr.space/ocrapi

[3] Wikipedia, Jun 2018. *Software Design Patterns.* [Online] https://en.wikipedia.org/wiki/Software_design_pattern