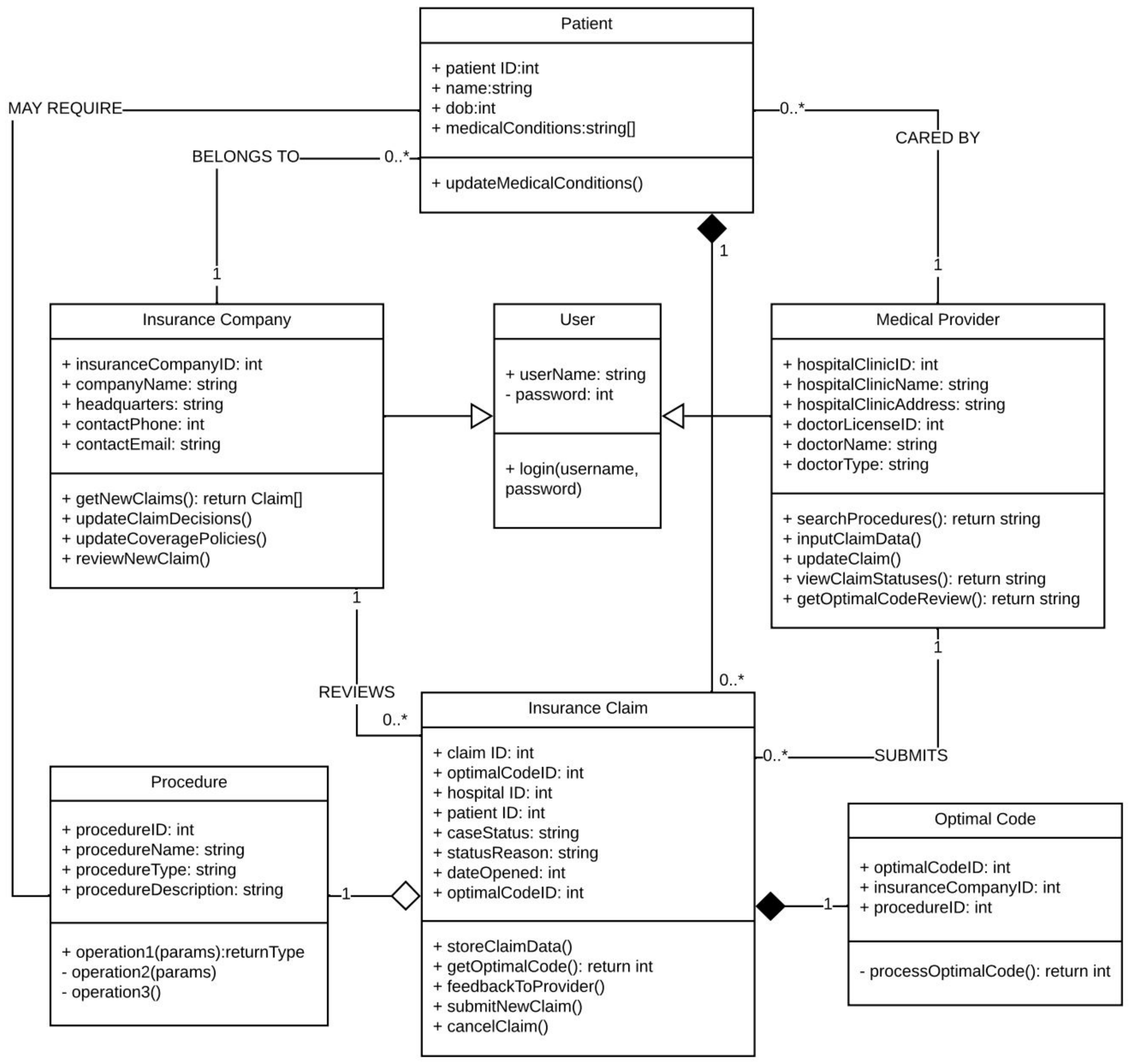
Homework 4
CS 361: Software Engineering 1
Group 7

Healthcare Coding System

Group 7: Homework 4

Justin Bethel
Jeffrey Visosky
Emmaniel Rovirosa
Jenny Rose Hanna
Joseph Stallano

UML Class Diagram



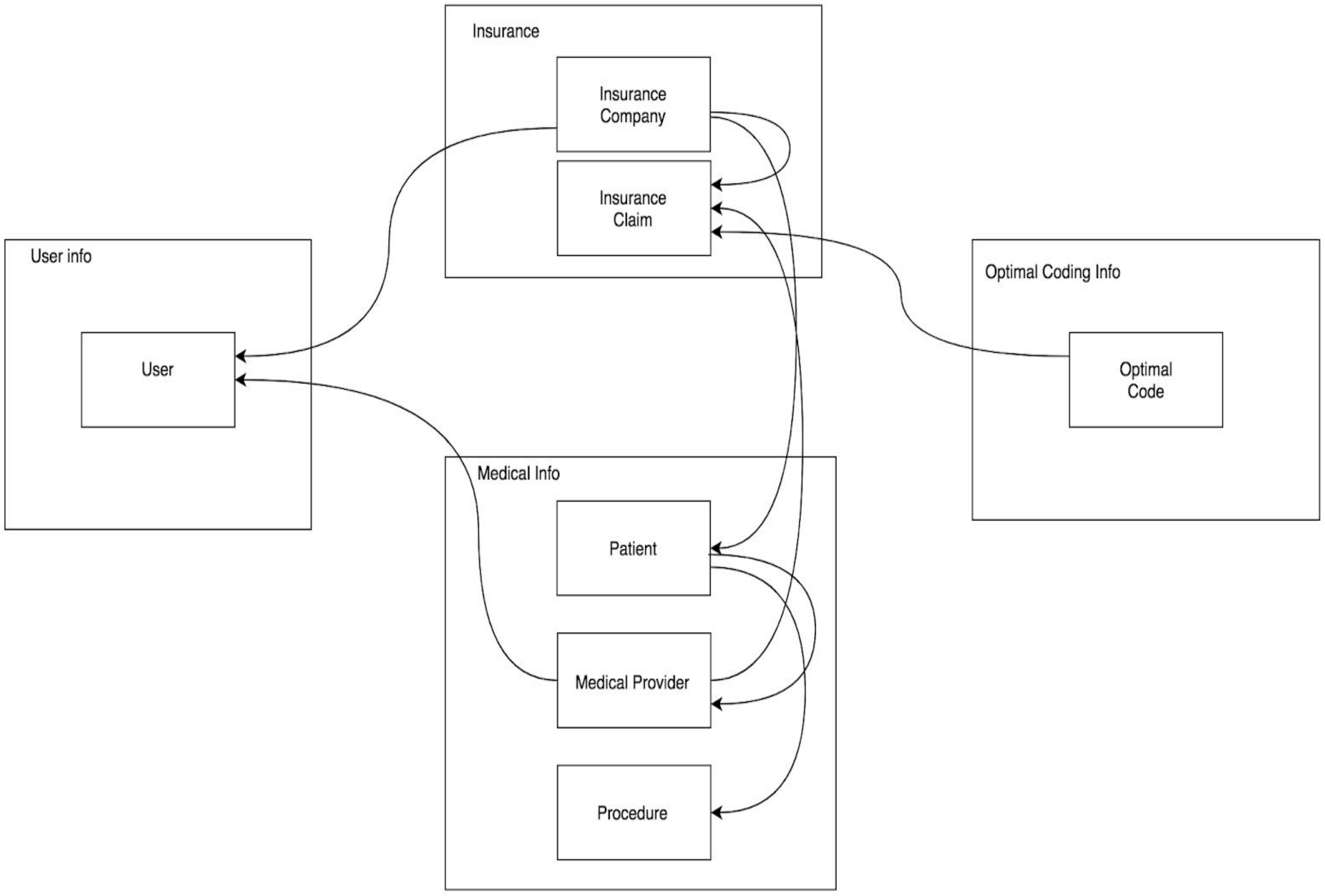
Incremental vs Iterative Development

Iterative Development - Iterative development requires building a small, working prototype of the system to get any value from the system and do meaningful testing. The prototype builds in size from there in multiple iterations of refinement and testing. In this case a basic prototype system in which simple claims can be sent and transformed with simple optimal codes. A very basic form of each feature with the entire system and classes (both user and client) will be built then tested, documented, and refined. This type of development would be useful in the cases that all the various components of the system require so much communication and dependability that this type of development is required.

Incremental Development - Incremental development requires building the entire system one step at a time in a fixed number of steps. According to the lectures, this is best used when one component of the system is involved in all others and one doesn't need an entire working prototype to begin accurately testing what's been developed. If we were to take this approach and build the system from the core outwards, we would first need to begin with the optimal coding functionality (generating optimum codes from input procedures and diagnoses) and storing these in the optimal coding database. This would truly be the core of our application as it is the primary feature of the application. This would be built to run on the server and receive diagnosis/services as input to query the database as well as policy updates to modify the database.. Then, the client-server networking ability would be added such that a client application can send procedures and diagnoses to the server and it will generate the codes. Then, the user systems (and classes such as Provider and Insurance company) will be created and the send claim with optimal code function will be implemented. After testing, the ability to modify and update claims as well as view them would be implemented using the claims database. Finally, the authorization and front-end systems would be finalized and tested.

Overall, I find that incremental development may be more useful in our case since developing the core feature of the application (translate services to optimal code) is the meat of the program and the largest, most challenging problem to solve. The rest of the features can be viewed as auxiliary to this primary function because they all depend on this function. As such, developing the optimal coding core and database and subsequently adding the auxiliary systems after testing and documenting of the coding system would be best.

Packaging - Coupling and Cohesion



Our goal in designing the system was to have the modules as loosely coupled as possible. We supported designing a system that was heavily cohesive, because this allows for easy maintenance over time. Below, we outline which entities are coupled and which entities are cohesive, and the degree to which they relate.

Data Coupling

- Medical provider submits insurance claim data to insurance claim

Control Coupling

- The medical provider signs in to the user interface, requiring access to the user information
- Insurance company signs in to the user interface, requiring access to the user information

Stamp Coupling

- Patient information is updated when they are cared for by a medical provider
- Insurance company needs structured data from insurance claim in order to review an insurance claim
- The optimal insurance code is returned to insurance claim in the form of structured data. This information is needed to update the insurance claim with the optimal code.

Data Coupling

- The medical provider provides unstructured procedure data to the insurance claim. The insurance claim will use the information to fill in the necessary fields to fulfill the claim data requirements

Cohesion

Functional Packages:

- **User Information -** The purpose of this module is to encapsulate all user related information. Users included anyone that can sign in to our system.
- **Insurance Information -** The purpose of this module is to encapsulate all insurance related information. This includes the insurance company and insurance claim classes.
- **Medical Information -** The purpose of this module is to contain all information related to medical records. Because the patient is the primary concern regarding medical records,

the Patient class is included here as well as the obvious Medical Provider and Procedure classes.

- **Optimal Coding Information -** Lastly, we have the optimal coding information package. This contains a single Optimal Code class which is information related to the optimal codes that will be used to update claims.

Design Patterns

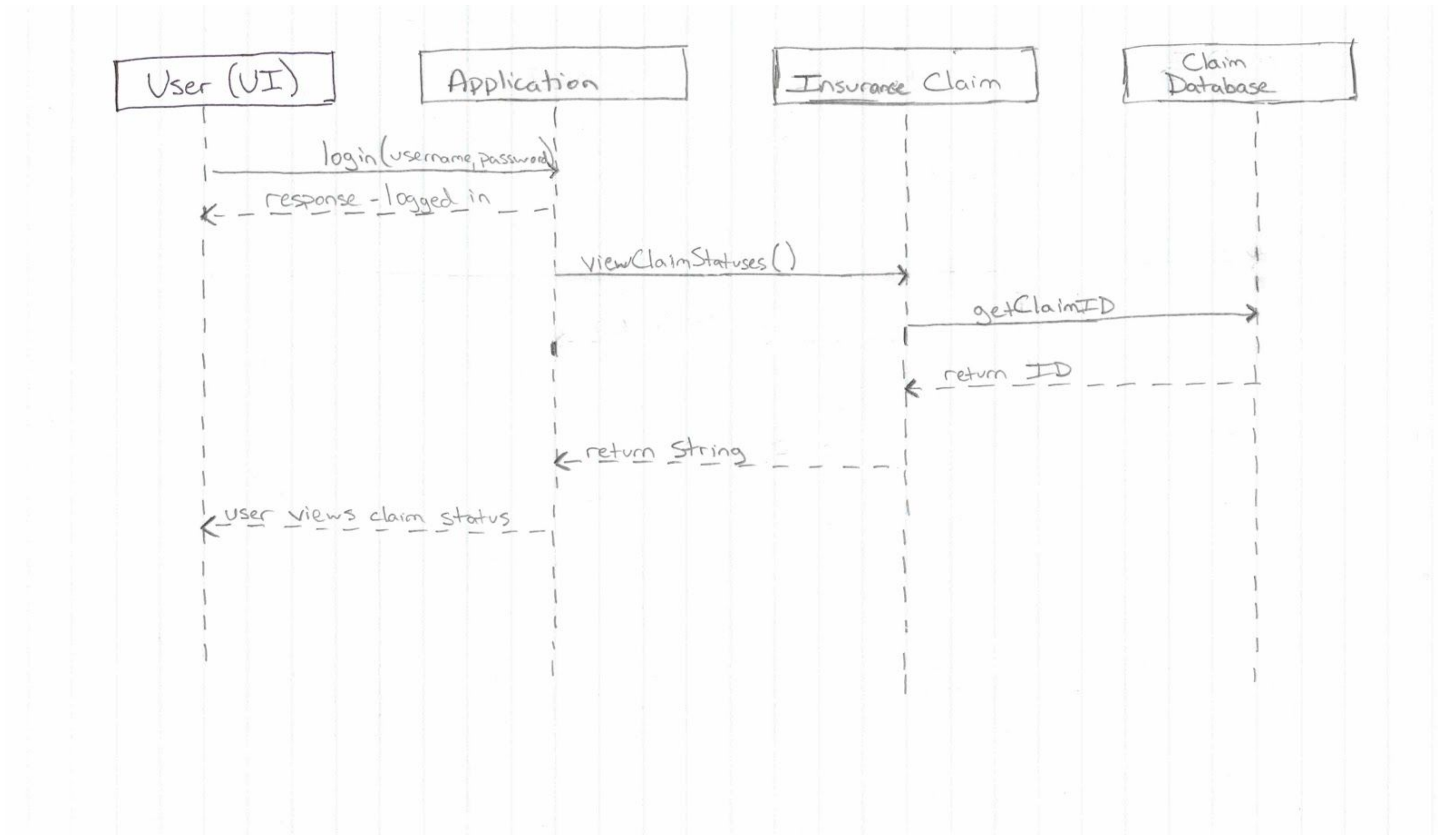
The main design pattern we will be using is the interpreter. The interpreter parses and acts on instructions written in a certain way. For our project we have to “adapt” what the doctors will input to the output that the insurance companies want to see. This would likely be in the form of making two interpreters. First, we would make an interpreter to adapt what the hospitals give us to the way we store information on our servers and databases. Then, we will have another interpreter that would then allow the information to be displayed in the way that insurance companies would like it. We need different interpreters due to the vast amount of hospitals and insurance companies which all require different things.

We could also use an observer that would observe the status of the case. The observer is a design pattern that watches for other objects to change state and then notifies. We could use the observer in our system by watching when the insurance company changes the insurance claim from say, “under review” to “accepted”. Once the observer senses the claim to change it will notify the hospital’s billing department. Then the staff can either update the claim for more information or mark it as complete if it was a favorable outcome.

We could also use the composite design element. The composite design allows for the reduction of complexity by dealing with a group of objects as a single object. This could be useful because it is likely that most hospitals and insurance companies will have many cases in our database. We can then group the data associated with each to do things all at the same time. For example, it is important to refresh the information that the clients have so that they are getting the most up to date information. This could be accomplished by saying that we are going to refresh everything at a hospital all at the same time instead of putting a timer of sorts on individual cases. This would also be useful for instances where an insurance company knows they won’t need the data often so they allow for the system to stop refreshing to their client systems.

Sequence Diagram

Use Case: I am the medical provider and I need to view the status of a previously submitted claim.



Interfaces

The interfaces in this system, besides the user interfaces, are between the systems that store and lookup different types of information. Here they are broken down by system, with inputs and outputs listed.

GUI (healthcare provider side)

Inputs: Takes in user info, patient information, and procedure information from the user; takes in response information from the Medical Information, Insurance Information, and User Information portions of the system

Outputs: Outputs user-generated information to the relevant system, and displays data received from other portions of the system to the user

GUI (insurance company side)

Inputs: Takes in user info, insurance code information, and claim status updates; takes in response information from the Optimal Code Information, Insurance Information, and User Information portions of the system

Outputs: Outputs user-generated information to the relevant system, and displays data received from other portions of the system to the user

Authenticator (User Information System)

Inputs: User info from the user interfaces, information from the User Database used for authentication (username/password)

Outputs: Status of authentication and user type, sent back to the GUI

Medical Information System Interfaces

Inputs: Patient information specified by the healthcare provider

Outputs: Previously entered patient information, sent back to the GUI; patient's insurance information, sent to the insurance claim system

Insurance Information System Interfaces (External)

Inputs: Takes in initial claim from medical provider, as well as updates to claim status from insurance company

Outputs: Returns claim status to GUI

Insurance Information System Interfaces (Internal)

Inputs: Patient's insurance information from Medical Information, procedure information from the medical provider, and insurance codes from Optimal Code Information

Outputs: A claim matching the procedure with the correct insurance code, returned to the medical provider for review

Optimal Code Information System Interfaces

Inputs: New and updated insurance codes given by the insurance provider

Outputs: Existing insurance codes, sent to the Insurance Information System; insurance code information returned to insurance provider

Fault Handling

Possible exceptions include:

- No code found for input diagnosis/services: In this case, a message would have to be sent back to the user that no code was found for a specific diagnosis or procedure so that they know the claim wasn't sent. Exceptions like this should be rare, considering the diagnosis/service form would be dropdown rather than input-text so every diagnosis or service/should have a corresponding optimal code. Exceptions like this would be more likely to occur if the database wasn't up to date (insurance policies incorrect) or the database was somehow modified in a way that corrupted it. Either way, the software maintenance team should be informed of this error automatically since it implies a broader problem with the optimal code controller and database rather than network or user error. A message would be sent back to the user that no correct code was found and to try again later. If the code continues not to work then they should call the support team.
- Claim not delivered to insurance: If an optimal code was found but the claim wasn't able to be delivered, this is most likely a network error in sending the claim from the server to the client. In this case, an error would have to be returned to the client (provider) and the error message would specify the error and ask the user to try again later.

- Unable to modify sent claim: This exception could be caused at multiple layers of the application which makes it more challenging. This could be caused by a failure to update the claim database or a failure to update the insurance company. The important thing here is that if there is a failure to update the claim database then the insurance company should not be notified about an updated claim. The vice versa is a more acceptable error - that being that the database could be modified but the customer was not notified. The claim controller could check to see if the customer received the notification and if not retry later for a limited number of attempts. It would most likely be wise to redesign the system to make sure the claim was updated in the db before a notification was sent to the insurance client.

Contributions

Joseph Stallano - Created implementation packages and assessed coupling and cohesion of the packages.
Justin Bethel - Covered incremental vs iterative development as well as fault-handling
Jenny Hanna - Interfaces
Emmanuel Rovirosa - Design patterns
Jeffrey Visosky - Built UML class diagram