CIDM6330 Project

Problem Identifiaction

In this project, the goal is to develop a local clinic management system, following Percival and Gregory's development process with some refactoring. The main development philosophy is to adhere to the dependency inversion principle to help us adapt our system more easily when business needs change in the future. The benefit of this is that it makes future maintenance and improvements easier and less time and cost consuming. The entire development process will adopt domain-driven, behavior-driven and test-driven design. The design pattern mainly involves six parts: domain, service layer, adapter, message bus, database and API.

In software development, the system is divided into multiple layers, and the central layer is called the business logic layer or domain. It contains the business concepts, rules, parameters, and results. The domain model should be defined before starting the efforts to create a database or program. Domain modeling is critical in system design as it helps developers and designers define the boundaries of a system and streamline software architecture. The primary goal of the local clinic management system is to provide users with the functionality to manage patient information and history, as well as make appointments. The central model in this project is Appointment, which ties together patient information and history, allowing the system to send notifications to users when profiles are created or changed. During the domain modeling process, it will be segmented into multiple models, including Patients, Patient history, and Appointment. The ideal business rule for the system is that when a patient makes an appointment, the system stores the record in the database and sends notifications to staff and patients. If the patient is new to the clinic, staff will create new patient information in the database and make an appointment. At the end of the appointment, the patient's history is updated in the database. After defining the model, UML (Unified Modeling Language) will be used to model the business relationship, and user stories will help project customers understand the system more effectively.

When the system model and diagrams are complete, the programming process begins. As mentioned earlier, the project will use test-driven design, which means tests will be written to ensure that the model and other features meet the goals. Although Percival and Gregory tried their best to decouple the system, their design patterns were only partially accepted, and this project required some refactoring. The main reason is that the project will use Django and Django Rest Framework instead of Flask and SQLAlchemy, which operate differently than Django. Because Django has an overall architecture that is more coupled than Flask and SQLAlchemy, following Percival and Gregory's design patterns will cause some problems.

Additionally, Django is monolithic and already has handlers, work units, repositories, etc., built into it. Therefore, some refactoring will be applied to Percival and Gregory's architectural patterns, such as repositories and units of work that will not be written. Since leaving the model code outside of the Django folder would make it difficult to get these suites to talk to each other and prone to test failures, this project would accept the monolithic structure of Django and would not write one's own repository and work unit implementations.

Once the design model for the project is finalized, the message bus will be taken into consideration. Django already includes a signaling feature that enables specific senders to notify recipients when certain operations occur. Since Django's monolithic architecture has already been chosen for the project, many parts of Percival and Gregory's design patterns will be replaced by Django. Therefore, Django channels and signals will be used for the message bus for the sake of simplicity and integrity.

According to the monolithic architecture of Django, it is more coupled than Flask/SQLAlchemy but has the benefit of simplifying development. Therefore, much of the discussion in Percival and Gregory's book will be superseded by the raw functionality of Django. However, relying heavily on Django and Rest Framework during the development process will also cause some problems.

1. Using Django Rest Framework's serializer and router to replace html.py declarations and cumbersome URL settings reduces a lot of work, but it is difficult to fix problems after they occur. In this project, the homepage and URL includes 'api-auth/login', and the user login and registration icons are not displayed when the "python manage.py runserver" command is executed. These issues should be solved by assigning 'urlpatterns = [path('api-auth/', include("rest_framework.urls", namespace="rest_framework")]' in urls.py. However, the home page is still not found, however, the home page is still not found, and api-auth/login will throw an exception of "rest_framework" is not a namespace registered and is raised within the file of "env/lib/python3.11/site-package/rest_framework/templates/login.html"

2. It is difficult to apply ForeignKey to model defining; for example, some errors will occur as the patient and appointment entities are tried to link together through the patient (name) attribute that points to the patient attribute. The request will fail, and a ValueError of 'Cannot assign "[<Patient: Harry Patrick>]": "Appointment.patient" must be a "Patient" instance' will be raised when the data is POST. In this project, the solution to this problem is to replace ForeignKey with CharField, but the relationship becomes manually constructed.

3. When the api folder is connected to the domain model folder, the attributes of the entity need to be repeatedly defined, and renaming the attribute names becomes cumbersome. However, this problem can be solved by executing unit tests.

In summary, adopting the design pattern for this project has several advantages. First, there are significant benefits to developing a system by dividing it into multiple layers, with each function located in specific folders and archives, which reduces confusion during development. Secondly, Django's monolithic architecture is quite suitable for beginners and professionals who want to develop systems because Django provides most of the functions. Third, test-driven and domain-driven development methods can help designers avoid future confusion as systems become more complex. These methods are implemented by defining the domain first to ensure the main functionality provided by the domain models. Tests are then written to ensure that the functionality actually meets requirements, which can also help developers find where errors occur.

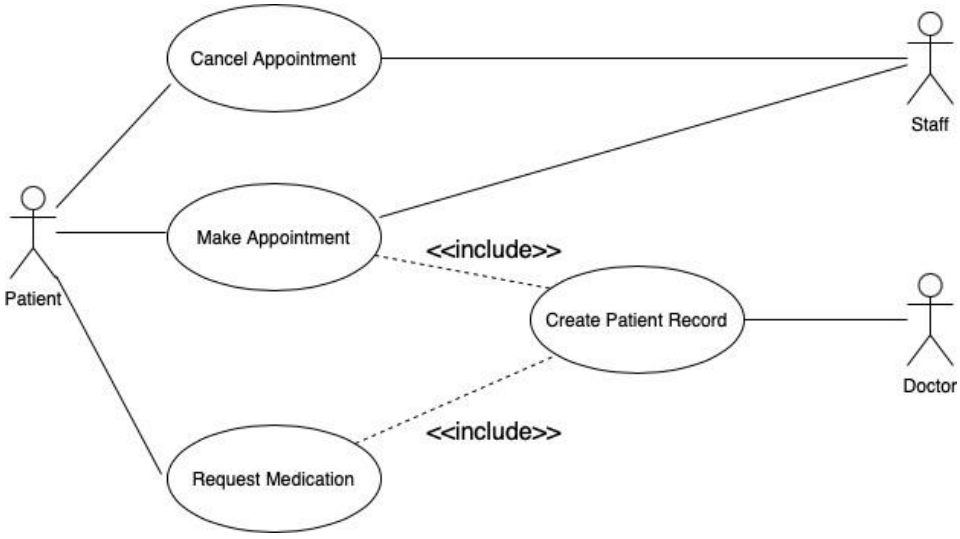
Use Modeling

1. User Story

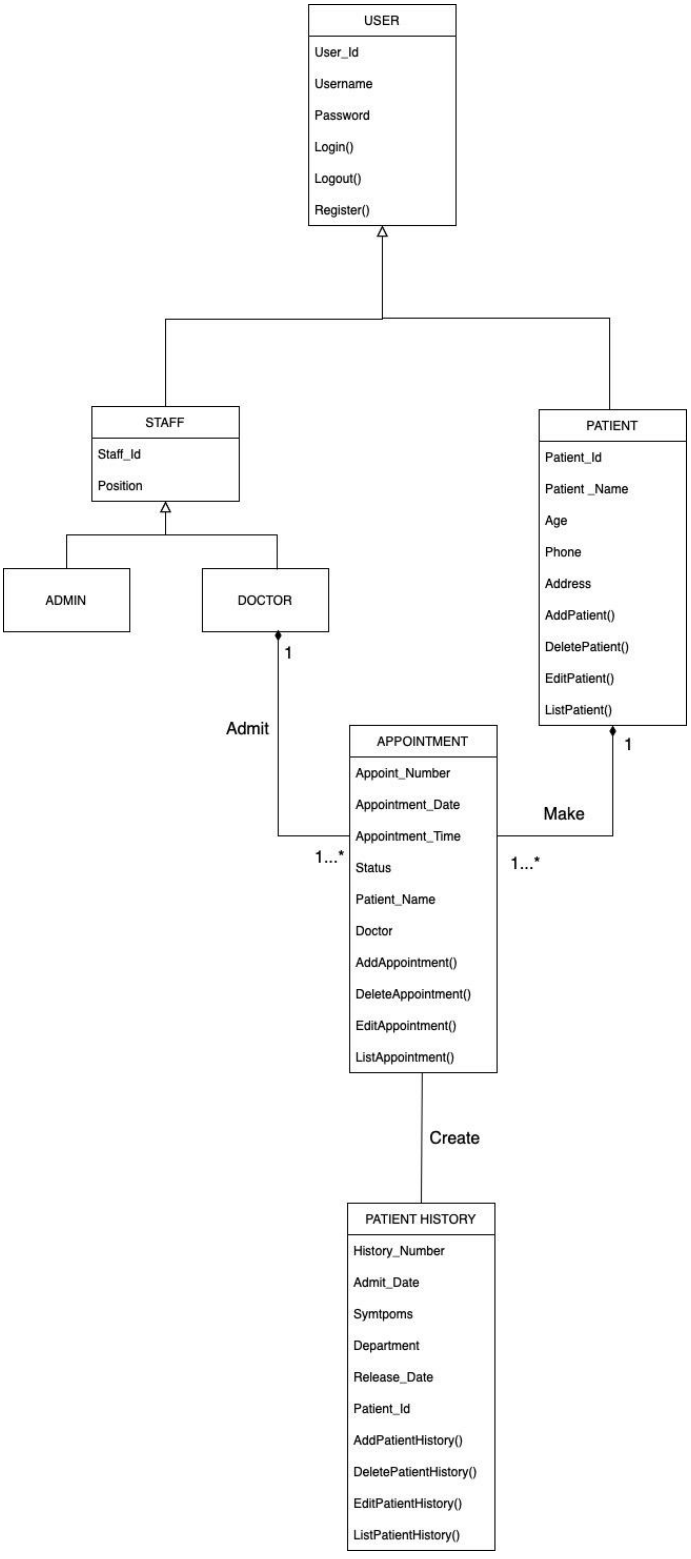
Maria works as a front desk clerk at a local medical clinic in Smithville, a village located in central Texas. Her primary duties include receiving patient appointments, scheduling those appointments, and editing patient information. Whenever a patient contacts the clinic, Maria has to ensure that both the patient and the doctor are available for the particular appointment date and time before booking the appointment for the patient. Additionally, if the patient is visiting the clinic for the first time, Maria must collect basic information about the patient upon arrival.

Jack is a doctor at the same medical clinic where Maria works. He receives a daily schedule of appointments from Maria. Whenever a patient arrives at the clinic, Jack reviews their previous medical history, diagnoses their condition, and records the new medical history in their medical records. Whenever the patient's information is edited, the system sends a notification to record all transactions that occurred, allowing the front desk clerk, physician, and pharmacist to understand the follow-up work.

2. Use Case Diagram



3. Class Diagram



Functional Test Plan

AppointmentTests, TestPatientCommands, TestPatientHistoryCommands, and TestAppointmentCommands. Most of the tests are similar; for model testing, the create, list, retrieve, delete, update, and filter functions are tested. First, define setUp using APIRequstFactory, then create an object for the test model, define the reverse URL required for later posts, and put actions. During the test definition, data input should include all properties defined in models.py. In each test, status_code is checked to ensure that the post and put commands were successful, which is the HTTP response success code, and then tested to see if the data in the repository is as expected. For command testing, the add, edit, delete, get, and list commands are tested. The data will be defined first in setUp, and when these are added to each command test, objects.count() and objects.get() will ensure that the functions work as expected.