Design Document #2

A list of design patterns that you used and why/how.

Preface: Our program uses several design patterns which allows it to be as extensible as it is. Some design patterns did not make sense in our code, hence why we did not use them. In this document, we justify our use of the design patterns we did use.

State Pattern

How it is implemented in our program:

- At a high level our program is essentially a finite state machine where each controller class is a unique state and is associated with a different menu or list of available commands that the user has access to.
- Each unique state has a limited number of transitions or ways to switch between states.
- We store the current state of the program in the `Context` class.
- The `Context` class runs the central loop of the program and provides methods for the changing state of the program while it is running.
- In each iteration of the central loop, the program delegates the task of handling a single user input to the current state.
- Each command the user inputs can act as a transition between states. For example the 'sign out' command represents the transition between the state that a user is signed in to the state where the user is not signed in.
- Each state class contains a reference to the `Context`, thereby allowing each state to change the current state, as well as to break out of the central while loop which corresponds to exiting the program.
- **Example:** If the program starts off in the sign in state (the state where the user is not signed in), the `Context` delegates to the current state (in this case `SignInController`) the task of processing a single user command. From here the `SignInController` can either remain in the current state (if a sign in fails for example), exit the program (by calling the exit from the `Context` within the `SignInController`) or change the current state to any user controllers (context can switch to any of the following: `PatientController`, `AdminController`, `DoctorController` or `SecretaryController` using the `Context` class' `changeController()` method from `SignInController`).
- The list of commands that are available to a user changes depending on the controller, which represents what menu screen they are currently interacting with.
- **Example:** when a doctor signs in, the program is in the state of `DoctorController` and the doctor has access to commands such as 'load patient' or 'view appointments'. If the doctor decides to load a patient, then the state changes to

`DoctorLoadedPatientController`, allowing the Doctor to have more commands/behaviors related to a specific patient (e.g. 'create prescription').

Classes involved:

- `Context`: the main context that stores the current state as well as contains the main while loop of the program. It provides a method for transitioning states to be called from within the current state/Controller as well as a method to break out of the main while loop of the program.
- `TerminalController`: the class all the states inherit from. It has an abstract method `allCommands()`, which is overridden in all child classes.
- All classes that inherit from `TerminalController`: these override `allCommands()`; this is what separates every state from the other. `allCommands()` contains all valid commands for that state. Then we access the relevant command and run it.

Why we implemented it:

- Using the state pattern allows us to get rid of the Switch Statements Object-Orientation Abusers code smell. This allows us to get rid of long sequences of if and switch statements that would initially handle which commands a user has access to. Furthermore, some commands require the selection of an entity to act upon thus adding further complexity to the sequence. In simpler words, without the state pattern, our code would have one class handling which commands are currently accessible to the user. Inside of that class there would be a `currentUser` variable and the method handling the available commands would contain many if statements that check the type of the `currentUser` and give appropriate commands back. Furthermore, in our program, some users can act upon other users (doctor gives patient prescription), which introduces one more class variable. More if statements would be needed to accommodate the new commands like when a doctor is acting upon a patient. This is obviously not extensible. If our code requires the creation of more if statements for each kind of interaction, something has clearly gone wrong.

- For example, when a secretary loads a patient, the state changes from `SecretaryController` to `SecretaryLoadedPatientController` thus when we call 'book appointment' we cannot encounter a situation where the patient is null because that command is only listed as a command when we are in the state of having a valid patient and a valid secretary.

- It made operations such as 'sign out' or 'back' almost trivial to implement. In a program with ten menus, having such a simple feature as going back to the previous menu would have been an absolute nightmare. With the state pattern it was a few lines of code.

- Thus, it made state transitions extremely easy by handing off control flow to a different class instead of nesting control flow within the same class.

Command Pattern

How it is implemented in our program:

- The main interface involved was the `Command` interface which supplies a single method called `execute()`.

- We have many different nested classes within the controllers implementing the `Command` interface.

- In all of our UI controllers (that inherit from the base class `TerminalController`) there is a dictionary from Strings to Commands called `allCommands()`. This is made possible by representing commands as classes that implement the `Command` interface and, as a result, we are able to store them within the dictionary. We can access the commands via the keys and run them using the `execute()` method that all commands share.

- Note as well that while these classes are nested within other classes, they are extremely easy to move off into their own class and any time. Whenever a single command was used by more then one controller, we moved it to a place that both controllers had access to. The main reason we didn't do this by default was because we felt that as long as a nested inner command class was only used by a single controller, it was the best place to leave it in terms of clarity and privacy, since it allowed us to make many previously public methods into private ones.

Classes involved:

- `Command` interface: the central interface every command has to implement which contains the `execute()` method for executing a command.

- `TerminalController`: has an abstract method called `allCommands()` as well as the method `processCommands()` that takes in a String from the user and processes it before querying `allCommands()` in order to get the command associated with that String and then running it.

- Every class that inherits from `TerminalController`: they all have the method `allCommands()` which contains a mapping from this controller's command names to command classes that implement the `Command` interface.

Why we implemented it:

- Different commands need access to different variables but there should be a unified way to call a command.

- It made it so that we didn't need to use a massive switch statement to handle all the different possible inputs a user could do. This made the code more readable and maintainable, partially because it enforced that each user input has to be associated with one single `Command` class.

- A large portion of our program is made possible by this design pattern as it allows us to easily scale to as many commands as we need without worrying about making things too complex.

Command pattern and state pattern interaction:

- The Command Pattern coupled with the State Pattern made an extremely powerful combination, essentially allowing us to easily add more and more features without refactoring our code or changing our system. We have ten unique menus in our program each with multiple ways of inputting commands (either via numbers or name of command), with each menu containing as much as fifteen commands. With these two patterns, adding a new menu was extremely simple:
    - Create a new state corresponding to that menu.
    - Add any of the dictionaries we want to inherit commands from to that state's dictionary.
    - Add any new commands as nested classes, then add them to the dictionary as well.

  This greatly reduces the code for the new state.

They follow SOLID for the following reasons:

- **Single Responsibility Principle.** Each state is only responsible for keeping track of what commands are in the dictionary. Then each command is only responsible for one activity. Because each Command is its own nested class, separating them is extremely easy (something we did many times while writing our code in seconds). The `Context` is only responsible for running the while loop, while processing the input is left to the `TerminalController` abstract class.

- **Open-Closed Principle.** To add a new menu, it's extremely simple. Just create a new state controller and add the commands you need to it. To add a new command just create a new class that implements the `Command` interface and add the command to the dictionary

returned when calling `allCommands()` in the menu by adding the following code: `commands.put("<command name>", <newCommand>).`
We don't need to modify existing code to add a new feature and thus we follow the Open-Closed Principle.

- **Liskov Substitution Principle.** Substituting a parent class with a child class does not remove from the public methods in the parent. Therefore the classes are substitutable and abide by the Liskov Substitution Principle.

Stream/Iterator Pattern

How it is implemented in our program:

- All of our databases that implemented `DataMapperGateway` allowed us to get an iterator/stream of all items in that database. Most of the processing within the use cases was done with this pattern.
- For example of storing references to the appointments in the doctor or patient, we would store all the appointments in the appointments database and if we needed all the appointments with a specific patient and doctor, we would simply call `appointmentData().stream().filter(x -> x.patientname == patient.name && x.doctorname == doctor.name)`
    - This iterates over the entire database and only selects the items that fulfill the condition.
    - This allowed a universal way to access all databases, as well as an easy way to add features to the program, since you don't need to modify anything that previously existed, instead you add a new database with the object you are adding.

Classes involved:

- All classes that implement `DataMapperGateway` as well as the common higher order function `getByCondition()`: They have the ability to get a stream and an iterator
- All of the use cases that interact with `DataMapperGateway`: They use `map()/filter()/getByCondition()` to process information in the database. Some examples of this can be found in `AppointmentManager`, `PrescriptionManager` and `LogManager`.

Why we implemented it:

- The main reason we implemented this is to avoid interacting with raw datatypes directly for example: a hashmap of id to entity.
- To keep code uniform. Most of our use case code is centered around interacting with the database. Thus we had much repetitive code where we had to access items in the database

with certain characteristics, such as getting all appointments between a specific patient and doctor.

- Using streams allowed an easy and universal way of getting information from the database and changed the way decided to store entities.
    - For example, initially we intended to store a reference to all appointments in patients and doctors. Once we started with the stream pattern it felt far more natural to store the appointments separately.
- We also felt that at a high level it was likely easier to transfer this current design to a database eventually, as the language of `map()`/`filter()` is very similar to how you access items from a database.