Tommy Hua
Pranav Subramanian
Project 4

Semester Project First Development Pass – *Mongoose, A Competitive Wordle Game*

Status Summary

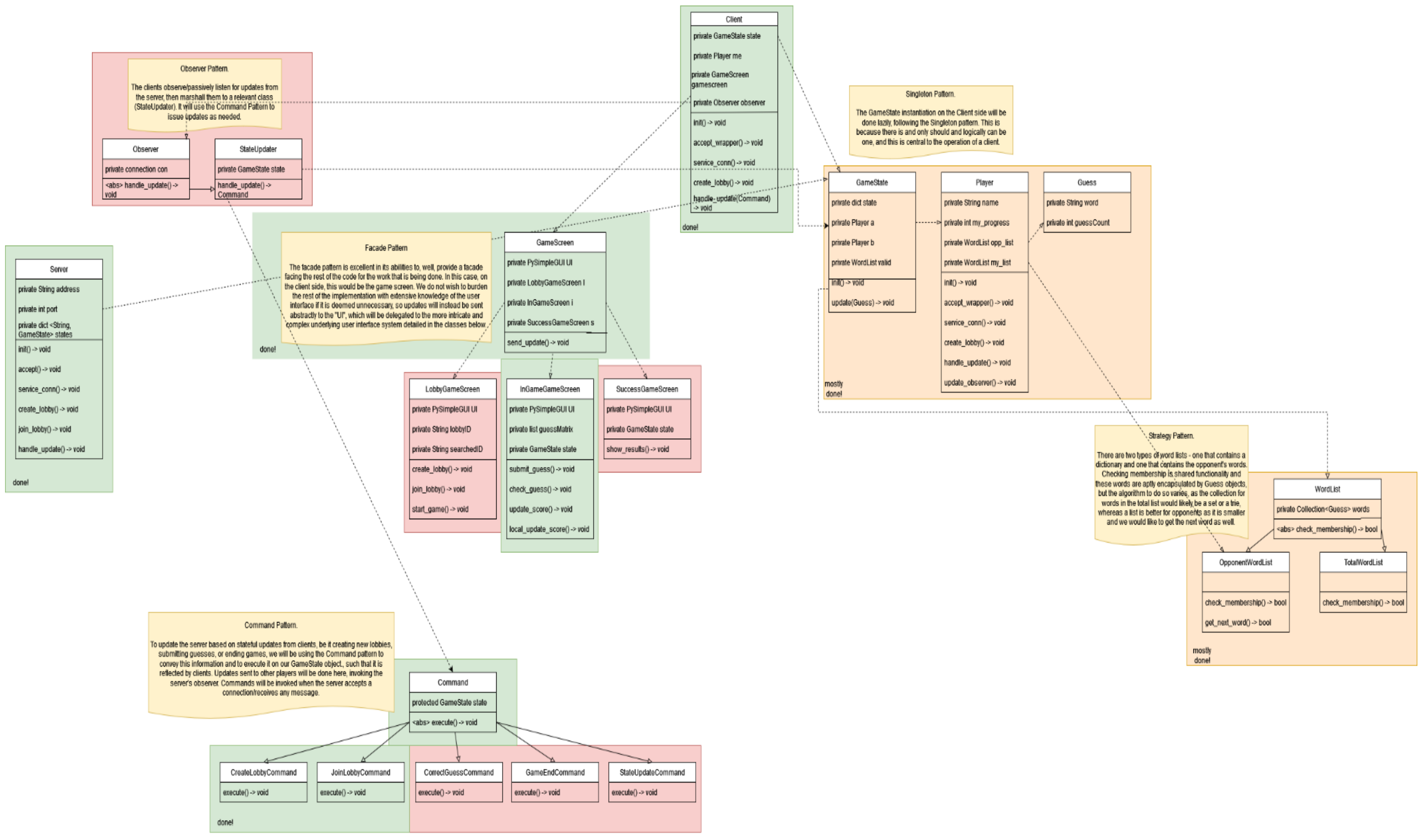
A lot of the work involved in this first sprint surrounded figuring out how to adapt our solution to Python. We made the decision during the last project as we designed our approach to switch to a different language - we were initially going to use Unity3D, but an object-oriented approach there, while very possible, seemed pretty suboptimal - we would have to stretch the requirements for various design patterns a *lot* to make it work, and it just wouldn't have panned out as well following the requirements for this assignment. Doing it more from scratch using Python, then, made more sense for the requirements of the class.

After we went through our design, got ideas on how to write object-oriented code in Python (beyond what was covered in lecture, i.e. putting it more concretely into practice than we had before), we got to implementing. One of the biggest challenges to implementation was figuring out both where to start and how to delegate work. We ended up delegating a lot of the backend to Pranav and a lot of the frontend to Tommy. While in our UML diagram this seems like Pranav would be handling a lot of server code, we proved to have delegated very effectively as the user interface, while only compromising a small number of the classes we would need to implement, ended up costing a lot of time as we were working with very foreign (and poorly documented) libraries.

Generally, during this first sprint, we achieved a lot of foundational groundwork that we can clearly expand on going into our second sprint. The functional components we addressed here were: **user interface**, **lobby system**, and part of the **word-list strategy pattern implementation**, which leaves the **game state management**, the rest of the **word-list strategy pattern implementation**, and **observer/server game state management**. This means we got most of the lobbying working (clients can connect to servers and join lobbies, servers can manage lobbies and the basic infrastructure of game state representation, and we have the general Wordle UI implemented, with just a little left for other screens and client integration).

Class Diagram

The annotated diagram follows:



Observer Pattern.

The clients observe/passively listen for updates from the server, then marshall them to a relevant class (StateUpdater). It will use the Command Pattern to issue updates as needed.

Facade Pattern

The facade pattern is excellent in its abilities to, well, provide a facade facing the rest of the code for the work that is being done. In this case, on the client side, this would be the game screen. We do not wish to burden the rest of the implementation with extensive knowledge of the user interface if it is deemed unnecessary, so updates will instead be sent abstractly to the "UI", which will be delegated to the more intricate and complex underlying user interface system detailed in the classes below.

Singleton Pattern.

The GameState instantiation on the Client side will be done lazily, following the Singleton pattern. This is because there is and only should and logically can be one, and this is central to the operation of a client.

Strategy Pattern.

There are two types of word lists - one that contains a dictionary and one that contains the opponent's words. Checking membership is shared functionality and these words are aptly encapsulated by Guess objects, but the algorithm to do so varies, as the collection for words in the total list would likely be a set or a trie, whereas a list is better for opponents as it is smaller and we would like to get the next word as well.

Command Pattern.

To update the server based on stateful updates from clients, be it creating new lobbies, submitting guesses, or ending games, we will be using the Command pattern to convey this information and to execute it on our GameState object, such that it is reflected by clients. Updates sent to other players will be done here, invoking the server's observer. Commands will be invoked when the server accepts a connection/receives any message.

Plan for Next Iteration

For the second sprint, we seek to finish the lobby system (there is a bit of code refactoring that can be done with it - functionality is complete), add components to the user interface, flesh out the rest of the gameplay system (this just entails game state management, linking it to our complete UI, and message passing, which isn't much more work beyond what we have laid out thus far), and write out our observer (which will be part of the above when it comes to message passing).

To do so, we will first integrate the user interface with our client and get that class polished. Following that/with that is a more thorough implementation of the WordList strategy pattern, which, while a little excessive, would pose an interesting problem and allows us to implement the strategy pattern in a fitting way - we will use more advanced search and general data structures in one list versus the other, and in doing this we can complete a lot of the client and game state updating functionality. Once we have done that, and added more to the user interface, we will move to implementing the observer pattern. This is crucial to message passing and therefore to overall game flow. When doing this we also need to handle the rest of the command implementations (as those are how we pass messages between client and server and general state communication). Following this comes testing, and completion of our project. Having developed and gained greater familiarity with our plan, project structure, problem space, and language, we don't anticipate the next sprint to be an infeasible load.