Tommy Hua
Pranav Subramanian
Project 7

Semester Project Final Writeup Submission

I. Project Overview

Project Name:

Mongoose, A Competitive Wordle Game

Team Members:

Tommy Hua (frontend), Pranav Subramanian (backend)

II. Final State of System

Our first sprint consisted mostly of reworking our design to accommodate instead using python, as it was discovered that usage of Unity3D as initially imagined would not be useful. As a result, what we started with for this sprint was mainly just groundwork, which we had to expand on - about an additional 50% of project work was incurred as a result, as anticipated (as this was the second half of the project/the second sprint). What we started with was an introductory implementation of the user interface, with some of the UI figured out, a basic implementation of the lobby system, **part of the word-list Strategy Pattern**. This means that we had clients/users that could join games, and the roots for the actual adversarial Wordle laid out, but actual implementations of game functionality like the maintenance of game state, gameplay, the **Observer Pattern** implementation, and much of the **Command Pattern** was yet to be implemented. *We used the same pattern of work delegation, where Tommy handled the frontend and Pranav handled the backend, which proved to incur very equal amounts of work and time spent by both parties.*

The status of the system as we reach the end of the second sprint addresses all of this. What we now have is a full Competitive Wordle/Mongoose implementation.

This means we have players, or clients, that can create or join game lobbies on the server. Once the game is started, the server notifies (via the complete **Command** and **Observer** Pattern

implementations, where commands encapsulate messages and the Observer Pattern consists of an *observer class which publishes updates* from the server to subscribers (in this case, a *subscriber* implementation that is *integrated with players/Clients*)) each client/player of the start of the new game.

Then, players have to construct lists of words that the opponent must guess, which involves checking word validity against a large dictionary, where membership checking of a word list is abstracted away and handled by a monolithic **Strategy Pattern** implementation.

Following this, these lists are forwarded to the server, then to other observers, and gameplay starts. Guesses are submitted, checked (following membership checks using, again, the **Strategy Pattern**, as there exist different strategies appropriate to check membership within a list of words depending on word collection size and implementation), and the local gamestate of a client/player is updated (the GameState representation is where we make use of the **Singleton Pattern**, owing to a significant and necessary one-to-one correspondence between players/clients and the state of the game they are in), and that update is propagated to the server and then other players.

Finally, when the game ends, all players are notified, and the results of the game are provided to the user. The game is implemented with two interfaces: a command line interface, and a complete but not completely coupled/integrated user interface, which abstracts away the intricacies of each screen in the user interface by a general UI facade (implementation of the **Facade Pattern**) which delegates updates to the relevant screen/user view.

III. Final Class Diagram and Comparison Statement

What follows now is the original UML diagram. Following that is the new UML diagram, with different design patterns highlighted, mirroring what was submitted in Project 5 (the design submission). Then, we will have a discussion of how we upgraded and redesigned as we implemented and encountered new issues from Project 5.

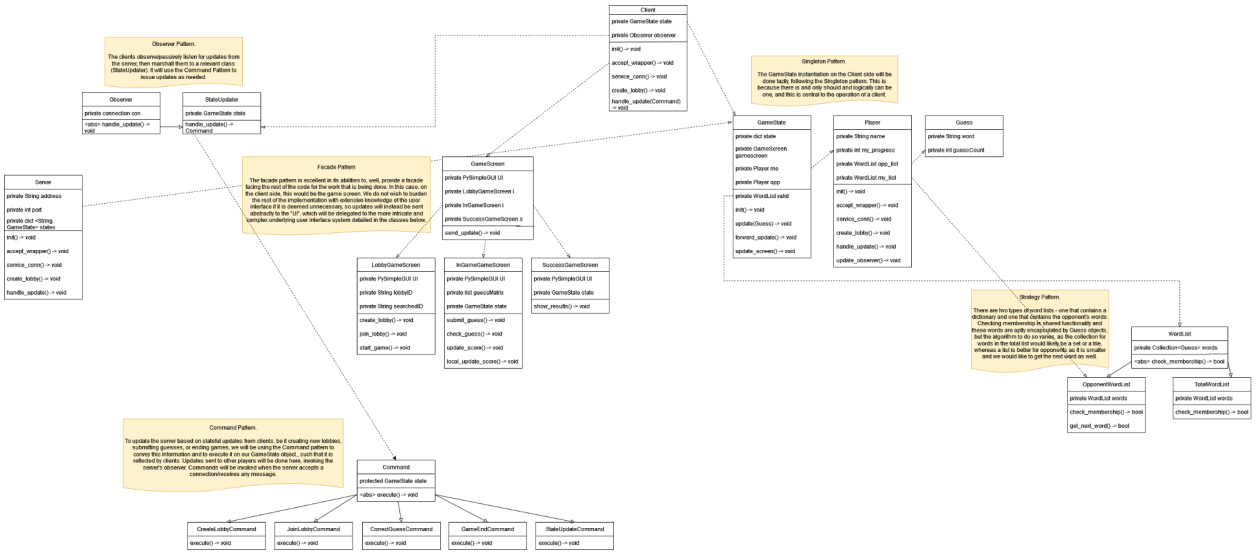

Figure 1: The old design.

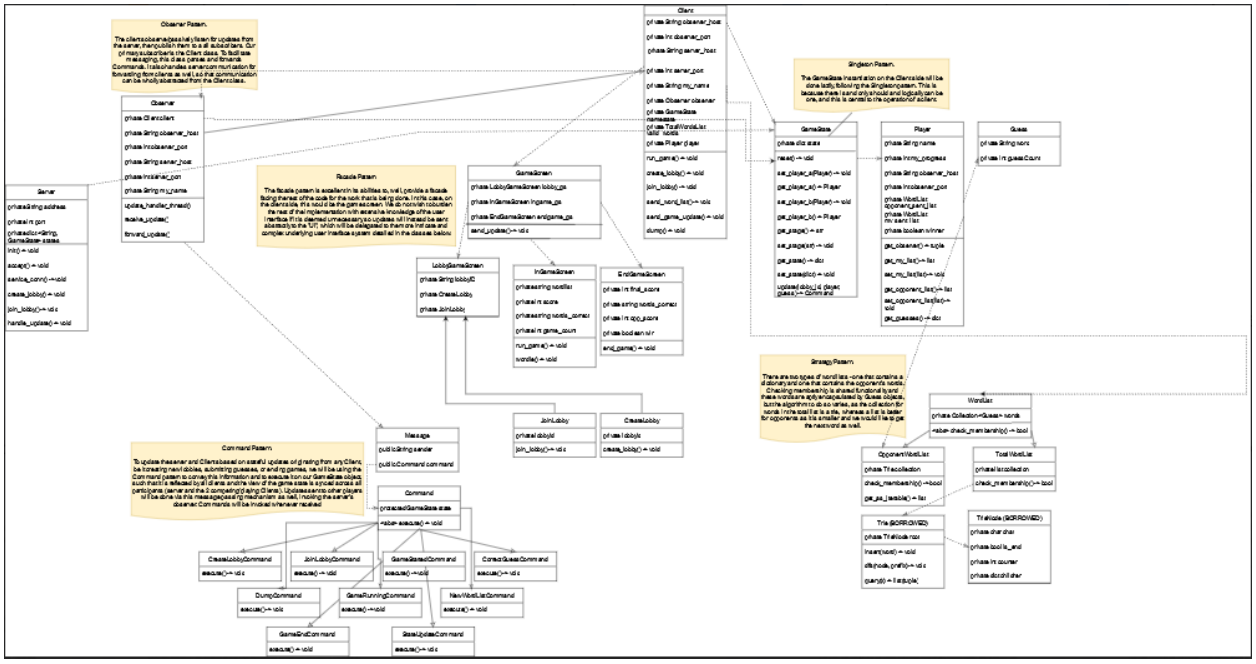

Figure 2: The new design.

So, here's generally what changed:

- The Observer Pattern was modified such that the actual subscriber was integrated into the Client. This is because the subscriber would literally just be receiving messages via function calls from the observer class/publisher class itself and then forward these/call relevant methods on the client. While, in an object-oriented fashion, this appears more correct, this is generally not what we desire, as it ended

up being an additional class and design bloat that we didn't wish to deal with. For ease of implementation, debugging, and overall readability and interpretability, we decided to merge that separate subscriber with the client itself, as the client was, effectively, a subscriber of the Observer class/publisher. *A separate subscriber implementation just to abstract it our, even though it only invoked methods on the client was redundant. The StateUpdater from before is now gone.*

- New commands were added, commands which weren't previously anticipated but proved to be very useful in implementing communication between Observers and Servers.
- Gamescreen coupling was not implemented entirely as the old design entailed, as while the Facade pattern does see (modified) completion as detailed later, full integration of the UI proved to introduce a large amount of novel problems that we could not reasonably address.
- New Client and Server methods were added to handle different state changes and types of updates, as they made implementation generally easier, and furthermore increased readability by breaking out small segments into their own methods.
- The handling of updates was done in part by the Observer, for parsing purposes, before being published to subscribers. The previous design had delegated much of this work to the GameState or exclusively the client, but as a lot of message parsing became necessary in the process, a decent amount of replicated state overhead was done in the Observer itself, prior to publishing of updates to subscribers.
- This is not a design change but worth noting - several of these classes were merged into the same files to prevent circular imports.
- A trie implementation was added to support the TotalWordList.
- A lot of functionality, such as lobby management and creation, GameState and user-facing updates, propagation of said updates, and the like was delegated more appropriately to the master of state management, the Client and the Observer (as well as the Server, which served as a replicator), as opposed to the objects themselves, like the Player and GameState, as these updates required views of data and information that only the Client and Observer classes have, so delegation to other classes/holders of data wouldn't make sense and would incur excessive overhead in passing information were update handling delegated to those classes..
- The Facade pattern was modified and implemented such that the three main game screens—LobbyGameScreen, EndGameScreen, and InGameScreen—are all handled by the facade class GameScreen. This facade class abstracts away the UI in order to not burden the client side with any unnecessary implementations. The screen classes are now made up of their respective UI implementations. In addition to this, LobbyGameScreen now implements two new UI classes,

CreateLobby and JoinLobby. These classes are the UI implementations of the create lobby and join lobby screens.

IV. Third-Party Code vs. Original Code Statement

There were several elements that were borrowed. What code was borrowed and citations/references are provided here (and in our comments). All other code is original.

The first was the implementation of the trie. We wished to use a more intricate data structure to illustrate and to best make use of the Strategy Pattern, and to do so we sought out a more fitting and intricate data structure that could be used for membership checks. This came in the form of a trie. So as not to expend implementation time on this data structure, however, as it is far from a focus of this assignment and any overarching object oriented principles, the implementation provided at https://albertauyeung.github.io/2020/06/15/python-trie.html/ was borrowed, used, and cited.

General insight and structure for having a multithreaded listener (both on the Server and Observer) came from https://www.geeksforgeeks.org/socket-programming-multi-threading-python/; some code is borrowed/structured similarly and cited in code in server.py and client.py. Random lobby id generation (single line in gamestate.py) came from here https://www.geeksforgeeks.org/python-generate-random-string-of-given-length/, and is also cited in the code.

The next was our user interface. The InGameScreen wordle gameplay functionality borrowed from the PySimpleGUI implementation of wordle. The foundation of the UI for the InGameScreen was derived from the PySimpleGUI but went under heavy modification and readjustment in order to serve our purposes and functionality. Link to GitHub: https://github.com/PySimpleGUI/PySimpleGUI/blob/master/DemoPrograms/Demo_Game_Wordle.py.

V. Statement on the OOAD Process

What follows now is a general statement on our experience with the object oriented approach and overarching process for this project. We will run through three different key design process elements/issues that we encountered in our analysis and design of this semester project:

1. **A huge challenge was in some of the abstractness of original design.** Our original design was done without an entirely perfect idea of how each component would be precisely implemented. This is typical in *any* design process, not just object oriented design and analysis, although since we did choose a heavily object-oriented approach, it is worth mentioning. *It wasn't entirely clear how the Observer pattern would be implemented, although it generally made sense to use it*, so figuring that and fleshing it

out further as we designed further and implemented our design revealed that there were more intricacies and difficulties than initially imagined.

2. **Some design patterns weren't totally optimal.** The general problem of multiple clients synchronizing state with a central server, which is what our game ultimately boils down to, doesn't necessitate an object-oriented approach, although this is one such route. As such, some of the design patterns used and other possible patterns did seem vaguely ill-fitted as a whole to this problem. For example, while it made sense in concept to do a full Observer pattern with a fully decoupled publisher, subscriber, and then a state maintainer that the subscriber reports to, it turned out that in implementation there was some overhead that this generic implementation introduced that a simplification which makes our implementation less dogmatically an Observer Pattern implementation proved to be easier to implement, understand, and work with. The same can be said for the Facade pattern, and other patterns. This could also be due in part to the fact that the majority of our experience implementing such systems is in systems-oriented languages, so a more elaborate design-oriented, object-oriented language did introduce some complexities. *Furthermore, object-oriented design and programming in Python, while supported, is generally a lot more awkward than Java, although server programming is generally more lightweight and easy to do.*
3. **Delegation of tasks to different objects proved to be a very useful byproduct of our design.** Chunking our project into different classes, and then abstracting away different functionalities by delegating work and decoupling functionality across different systems, for example making the Client, Observer, and Server try not to directly touch the GameState or Players, or making WordList interactions entirely contained within WordList implementations, helped chunk our problem into more approachable segments. By breaking things up, debugging and code readability also increased generally, as it was easy to follow what functionality was being invoked where, and who was responsible for what. While this did at first end up expanding our solution into many different classes and different places, which was intimidating, it ultimately led to a simpler approach.