Tyler Behm
John Chamberlin
Josh Killinger

2D Roguelike Game: Final Report

1. List the features you implemented (from part 1 and 2 where you listed the features you were planning to implement/requirements you listed). Show Final class diagram, discuss benefits of designing before coding.

Our original deliverables from part 1 were as follows:

Functionality:

- Player can move main character
- Enemies attack player
- Dungeon is generated
- 2D sprites visualize the game actions

[OPTIONAL] Stretch Functionality:

- Add more enemies
- Add more items

We implemented the functionalities, even the stretch goals. Notable features given below:

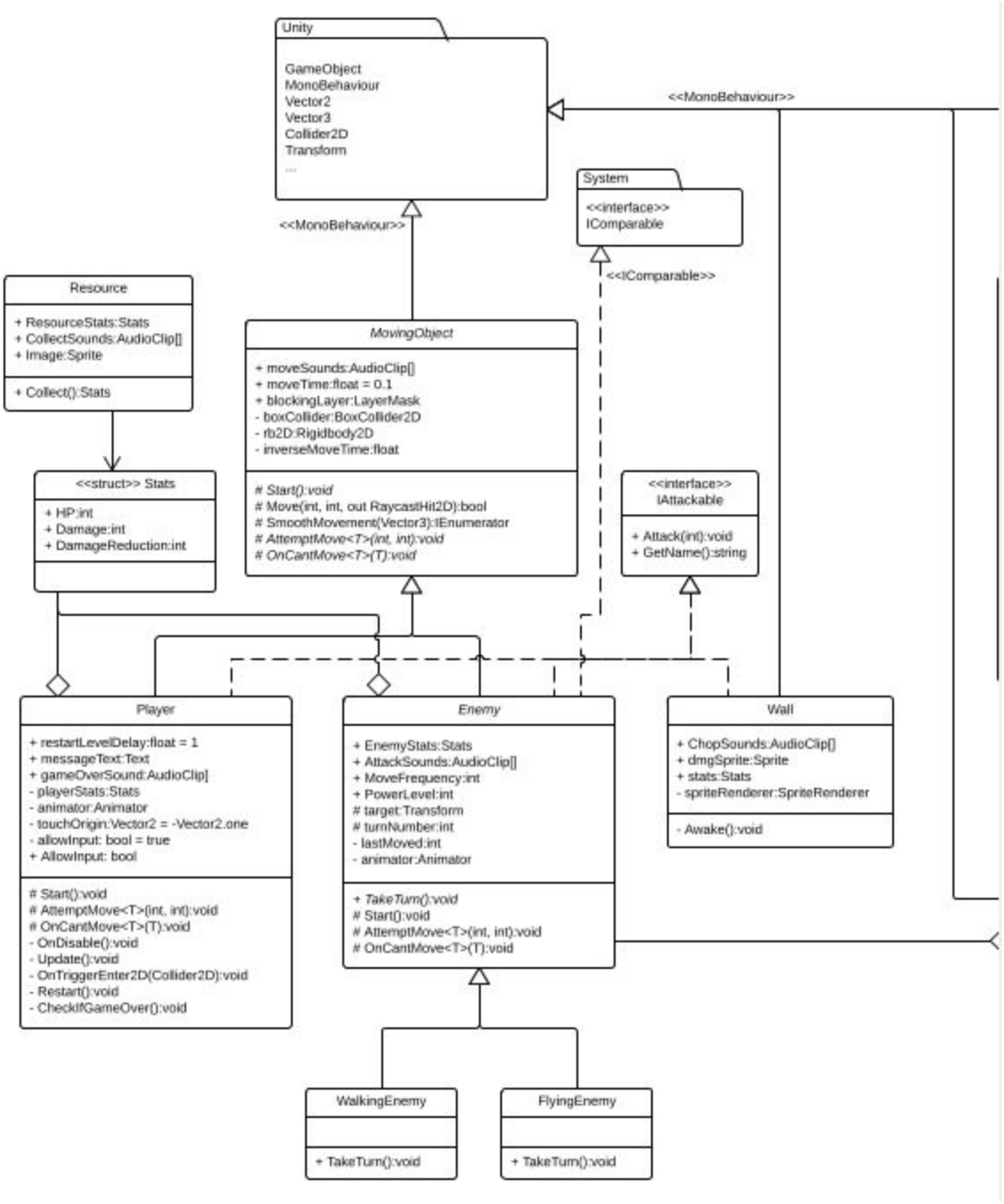
Added start menu
Added high score saving ability that persists data
Player can now attack enemies
Added weapons and armor for the player to pick up
Generated enemies based on power levels
Added new enemy class (flyingenemy)
Added a Mario themed skin to game

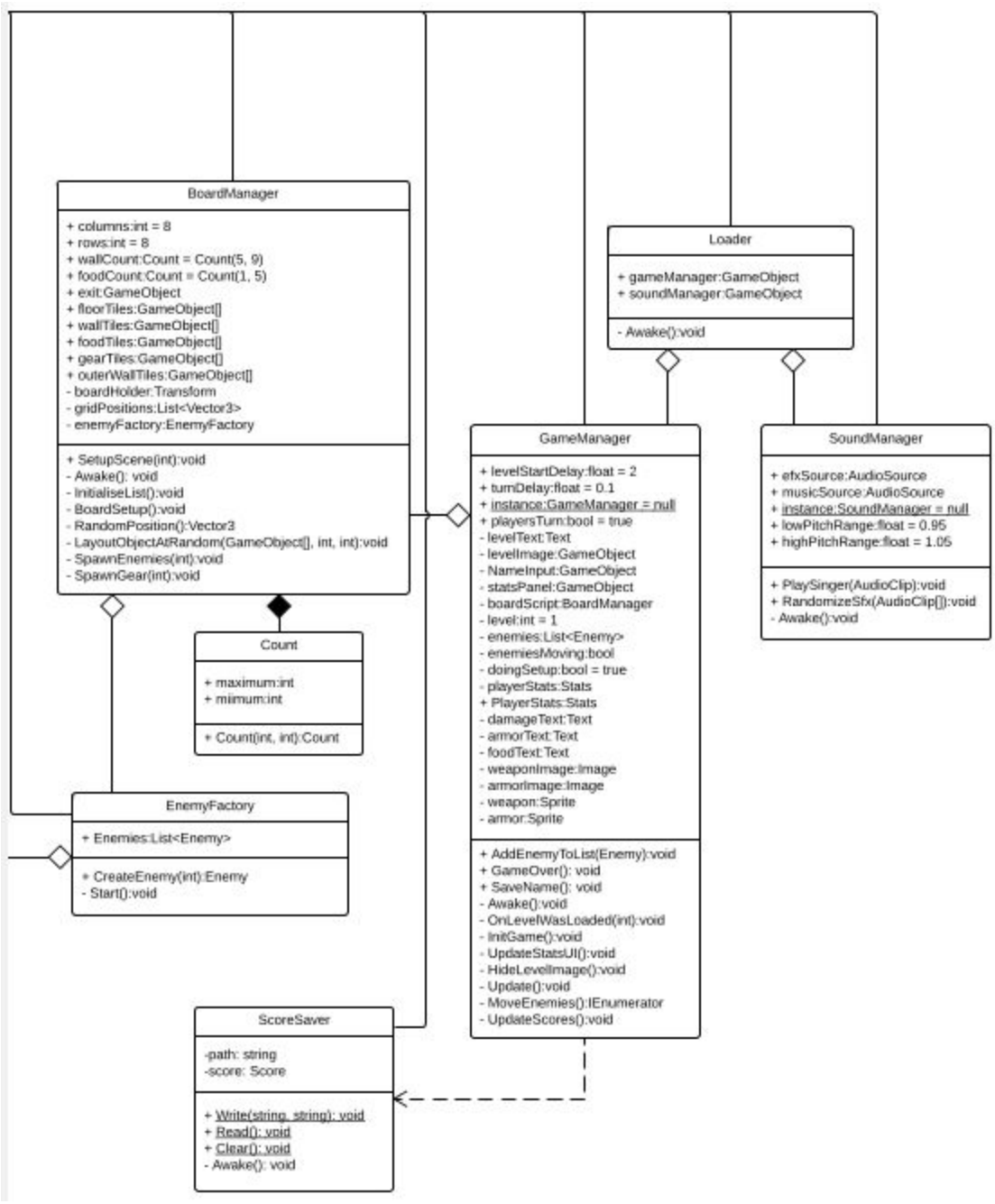
Designing before coding helped immensely with the development! The design laid out the roadmap which we needed to follow. Development was simply a matter of translating our diagrams into code.

Design made the development more organized. We never had to wonder what to do. And everybody had already agreed on what to implement so teamwork was consistent with the same plan.

Teamwork was more effective. While Josh worked on creating the enemy factory, Tyler was able to make more enemies. While Tyler worked on the score saver class, John was able to make the gameover screen for saving scores. While John worked on the in-game menu, Josh

was able to modify the game mechanics. Because we worked with the same design, our collaboration coordinated much better than without a design.





2. Design patterns

We used the **factory design pattern** for enemies. This greatly simplified level generation and the addition of enemies. We were able to give a higher power level to our enemy factory for higher levels so that it would randomly generate stronger enemies. Adding enemies to our game only required adding it to our enemy factory list.

We almost used the **strategy design pattern** for the selection of enemy movement patterns. We used inheritance, but we could have easily refactored to strategy. By making two enemy classes, WalkingEnemy and FlyingEnemy, we were able to decide the behavior of the TakeTurn() method at run time. Because of Liskov Substitution Principle, we can still generate either a flying enemy or a walking enemy from our enemy factory.

Component is a game design pattern which was not present in Gang of Four. The Mario-themed reskin was made easier by the **component design pattern** that was built into Unity. We did not have to remake the whole enemy to reskin. Changing the theme simply meant changing the animation component.

All of our enemies utilize the **prototype design pattern** via Unity's prefab framework. All enemy instances are either WalkingEnemy or FlyingEnemy. But they vary in properties like move frequency and damage because of the prototype prefab.

In the original code, the data for food was in the player class. This is bad coupling and is in violation of the Single Responsibility Principle. So we refactored food into its own class.

We made a Collectable interface for the food, weapons, and armor to implement. We made an Attackable interface for enemies, player, and walls to implement. This allowed us to reuse similar abstract traits but still encapsulate what varies in the classes.

We refactored player and enemy health points by introducing a parameter objects, Stats, which can then be composed into player and enemy. We did this because of its "duplicate code" bad code smell.

Score saver used data encapsulation to collect the name and level fields within an object.

For future work, we would implement the **observer design pattern**. At the moment, classes like player can print text directly to the screen. This is coupling that we don't want. With the observer pattern, we could separate the functionality. The player would instead trigger an event which would be handled by the UI.

3. Compare your Part 2 class diagram and your final class diagram - include part 2 class diagram and point out the necessary changes. You can answer why you didn't realize you needed things during the design and/or how next time you will be able to make a better design after this learning experience.

Our final class diagram changed a bit from the part 2 class diagram. But it still remained true to the original intent.

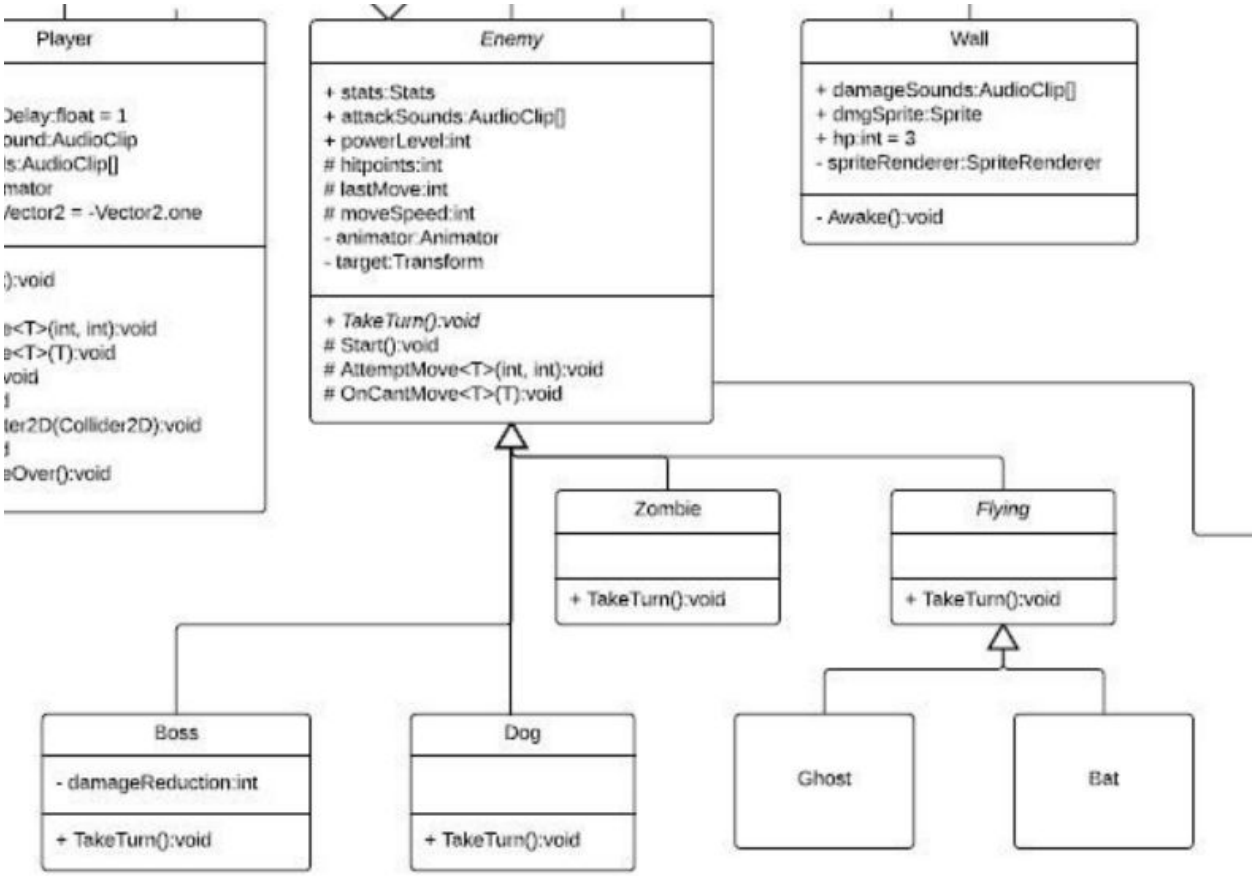
We moved around the enemy classes because we felt that it was better for gameplay. We made the boss class a subclass of flying so that it could attack and move diagonally. We felt that this

made the boss a more appropriate difficulty for the player. We couldn't have realized this at design time because we need to play the game.

Upon implementing the enemy functionalities, we found that it was easier to considering walking and flying enemies separately by using the strategy design pattern. This was a better way to implement it.

We had to wait until we could play our game to decide what power levels to assign the enemies. The enemy factory used the power levels to determine when to generate an enemy. If we made the power levels too high, we would not see them in our demo. But if we made the power levels too low, we would see the strong enemies too early in our demo. So we adjusted it by feel once we had a working demo.

For comparison, here is a subset of our initial class diagram for enemies:



4. What did you learn about analysis and design?

We learned that analysis and design is incredibly helpful for project development!

We learned not to rush into development because time is much better spent in design at first. We saw first hand how days of design saved us weeks of development. We saw how it greatly reduced the complexity of code to implement in our project.

We learned that design greatly increases the visibility of our code. It's hard to look at code and see the big picture. But by looking at our class diagrams, design patterns in our project became self-evident. By asking ourselves "where can I use this design pattern?" or "what object-oriented design principles apply here?", we were able to refactor our code to make it cleaner and more expandible.

We learned how analysis helps communication between customers and vendors and between developers within the project. Dr. Boese was effectively our customer in this project. By making use case documents, functional requirements, and activity diagrams, we were able to effectively communicate to her what were our project deliverables. Amongst ourselves, we were able to more effectively provide those deliverables because of the plan we laid out in our class diagrams.

Thank you very much for this learning opportunity, Dr. Boese.