Pokémon Showdown: Design Commentary (Fletcher Henneman)

Our goal was to create a working Pokemon battle simulation, similar to the online game Pokemon Showdown. We wanted users to be able to use their knowledge of matchups, move types, and more to increase the skill ceiling of our game. Making this game was very difficult as it required me to learn new libraries to fulfill our goal.

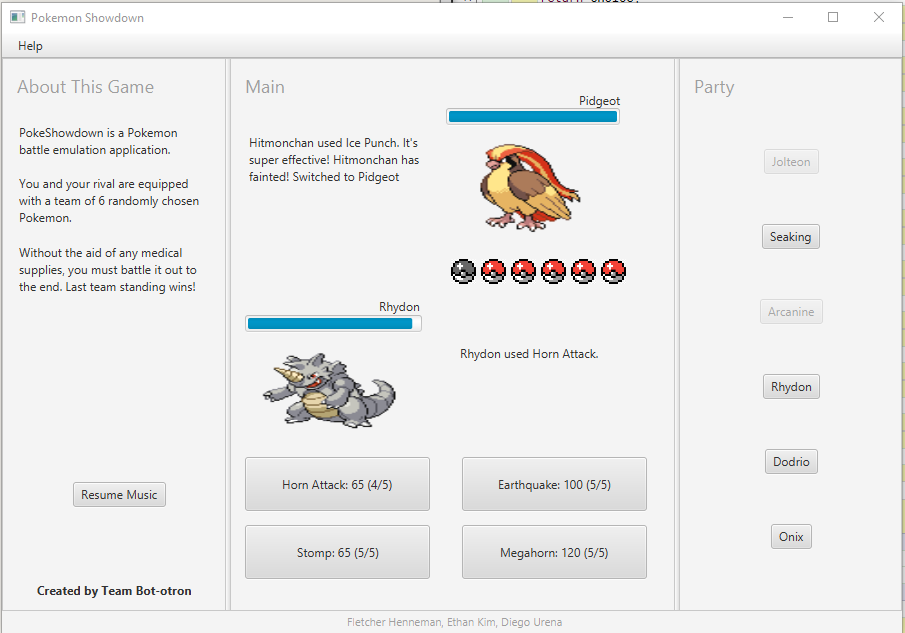
One such library was JavaFX. This library would allow us to create a GUI which was a colossal improvement from a text based game. In a text based game it would have simply stated what Pokemon were “on the field” and their health as well as giving you the option of typing a number corresponding to a possibility in the game. However, with JavaFX our final product was a user interface that looks like an actual game. However, no one in our group had ever used JavaFX before, or at least to any extent as close as this one. I was tasked with creating the UI so I had to learn this library from scratch as well as endure the barrage of errors and crashes that arose subsequently.

I began the project by first creating the UI. This gave me a better idea of what sorts of classes and methods we would need in order to serve how we envisioned the game. The first two classes created were UI and UIController. The UI class instantiated, setup, and displayed the UI window when the program started. The UIController class is what actually interacted with the UI. It could update UI elements such as text, play music, and receive events from when the user performed a specific action such as selecting a move to use. This became somewhat troublesome. As we wanted to add more features or change how classes could interact with the UI, I had to adjust the code accordingly. For example, when we created a Battle class which would handle the actual “battle” between the user and the computer, it would need to be able to change elements on the screen. To make this possible I had to have the UIController class pass the instance of itself as a parameter to the methods it called in Battle. This allows the Battle class to call the instance’s methods such as updating a Pokemon’s image, health, etc.

Next up was figuring out a way to import the existing list of Pokemon along with their stats and moves. This required us to make a Pokemon and Move class. Another problem we ran into was how close we wanted to stay to the original Pokemon games. We wanted to incorporate Pokemon types and Move types where certain types would be super effective against others (Water super effective against Fire). Then, when we wanted to calculate how much damage a move would inflict upon the opponent, we had to check the types of the user and the AI, incorporate extra “buffs” if one Pokemon was a higher level than the other, etc. This required a lot of research and painstaking coding as there was no simple way to implement it other than writing dozens of if statements (or that we could come up with at the time) which was all done by Diego and Ethan. Now, back to importing - I found JSON files containing 151 Pokemon, their base statistics (health, attack, defense, etc) and a list of their moves (damage, type, etc). To import them and deal with the JSON format I used the library json-simple (hosted by Google). Learning to use this was also a problem as I encountered errors which took hours to solve: Some Pokemon would not import and sometimes different Pokemon were missing information so our program would crash as it could not handle that.

Another obstacle was the handling of turns. Both the user and the computer had to come to a decision of whether not they should attack or switch out to another Pokemon. I had to code multiple cases such as then the user wanted to attack but the AI wanted to switch out (which would happen first). The original method of doing it was very complicated, inefficient, and was riddled with bugs that emerged only after completing its code. I was not content with this so I decided to redo it from scratch. In my planning I ended up deciding on creating another object to represent the data needed to handle this need, the Decision object. This object stored an enum depicting what type of action they were doing, along the vital information needed to carry out their desired effect. This made our task easier to handle, implement, and drastically reduced the amount of bugs that arose. One significant issue I ran into however was what if the user killed the AI before it attacked back and how to handle the effect of dying mid-turn and display proper messages. While we thought this would be a simple task, we had only thought of it at the end though. So we had to come up with a bunch of workarounds that did not require us to recreate many of the core functions of our game from scratch.

One thing we decided to change was how the user knew what actually happened. Previously we thought it would be easy to interpret but at the end we changed our minds. We now wanted to display a description next to the user and AI describing what they did that turn. Here is one example:



The Decision class simplified the process of adding this functionality as it contained all of the necessary information to come up with these descriptions. Again though, compromises were made as I hade to make two new methods to facilitate all the new possibilities (setExtra() and overrideMessage()). These new methods are not ideal but the program works as intended!