Jason Corriveau
Eric Marshall
Ben Matase
Alexander Murph

Design Manual

Introduction

We created a GUI driven Pokemon clone with a focus on the Battle functionality of the original games. By using sprites and databases from outside sources, we were able to give the game a realistic feel. Since the databases were in the form of XML files, we used the jDom library to parse and populate information into a form usable by our main code. Although not our original intention, we only followed MVC loosely; a lot of our controller logic was held in our view classes. For handling all of the nitty gritty game logic such as game states, frame rate, and drawing objects, we use the popular Slick2D library. One can see evidence of object oriented design in our Pokemon and Move objects. The Pokemon objects hold that Pokemon's unique stats, while the Move objects hold information about that unique move.

The next abstraction, the Trainer class, is an abstract class that is extended by UserTrainer and EnemyTrainer. All Trainers have a list of one to six Pokemon objects. A utility class called PokemonLoaderUtility fills in all the necessary information for a Pokemon given the species and nickname, then returns a completed Pokemon object. PokemonLoaderUtility also can fill in information for a move in a similar manner, returning a Move object. The GUI interacts with the controller through Events. Events is an abstract class that is extended by many specific events such as TextOutputEvents. The BattleController sends generated Events to the GUI where the GUI places them in an Event queue and the GUI procedurally executes the Events one at a time until there are none left. The BattleController simulates an entire battle between two trainers by calling methods in BattleSimulator: which simulates one battle sequence with two Pokemon attacking. The damage calculations are done in BattleCalculator. In the GUI, there is a ButtonManager which creates a two dimensional array to hold buttons and allow the buttons to talk to each other so that keyboard input is handled correctly so that the highlighting of buttons is correctly.

User Stories

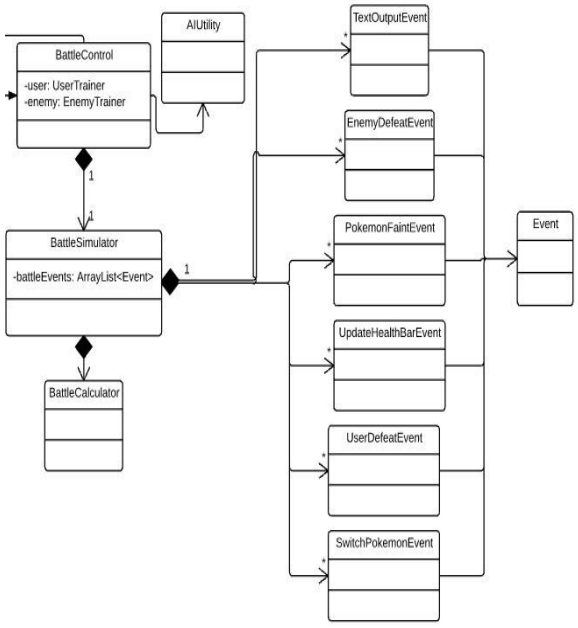
The following user stores have been completed in the process of developing this final project:

- As a player, I want to have a Pokemon battle GUI, so that I can battle Pokemon.
- As a player, I want a variety of pokemon options that work in the battle GUI, so that I can have fun using different Pokemon.

- As a player, I want trainers to battle and have them named after CS professors, so that I can battle professors like gym leaders.
- As a player, I want the ability to choose starting Pokemon from a database of Pokemon, so that I can choose my favorites and personalize the game.
- As a player, I want transition animations, so that the game feels like the real game and makes it look good.
- As a player, I want an elite four, so that I can test my skills and have an end game.
- As a player, I want music and sound effects in the game, so that I can get immersed in the world.
- As a player, I want randomized trainer battles, so that there are battles other than the gym leaders, etc.
- As a player, I want custom trainer art for the professors, so that I can see the differences between professors and see the Bucknell theme.
- As a player, I want the ability to lose/faint, so that I can be challenged by the game.
- As a player, I want unique moves for my Pokemon, so that I can develop strategies and make my Pokemon personalized.
- As a player, I want AI for all non-player trainers, so that I can be challenged when battling them.
- As a player, I want the game to be balanced, so that the game is fun but challenging.

The system that we have created fulfills all of these user stories. First, we have implemented a Pokemon battle GUI, as that is the core of our game. Then, upon starting up the game, a Pokemon chooser comes up, which allows users to select unique Pokemon and moves. After the user selects their Pokmeon, they are presented with a menu that allows them to fight either preset CS professors (and eventually the Elite Four is they get through the professors) or random trainers. Finally, once they enter a battle, there are basic animations (similar to the original Pokemon game) and sound effects that play depending upon what happens. The trainers (user and enemy) have trainer art, which is customized for each of the computer science professors and the elite four. Lastly, in the battles, the AI follow an algorithm to choose moves in a weighted random fashion, and the player can lose, as the game has been balance tested. Overall, throughout the game, sound plays to keep the game feeling immersive.

Battle Abstraction



The abstraction for a Pokemon Battle has three levels: Calculator, Simulator, and Control. Each uses the previous almost exclusively. In our BattleCalculator, we do all of the calculations necessary for a single Pokemon attack. That is, BattleCalculator is an object that takes in a defending pokemon, an attacking pokemon, and an attack. When we call the getDamage method on the BattleCalculator from the BattleSimulator, our Calculator returns how much damage that attack did to that specific Pokemon. It also takes into affect Critical Hits, which gives each move a small chance to do 50% more damage, and Accuracy, which gives certain moves a chance to do no damage.

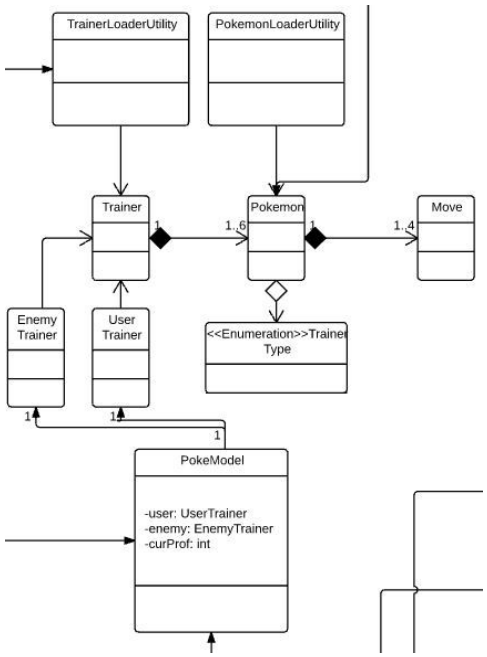
On the next level, we have our BattleSimulator. The Simulator takes care of an entire round in a Pokemon Battle. So, it takes in two pokemon and two moves, and waits for the Controller to ask for it to simulate the battle. The Simulator takes into account the speeds of Pokemon when deciding which one moves first, then go through the battle step by step and returns what happened within the round. The Simulator is capable of handling a user's Pokemon switching out and when either of the Pokemon faints. It also uses the damage information from the BattleCalculator to update the Model to reflect the current health of either Pokemon participating in the battle.

On our top level, we have our BattleController. The Controller holds a UserTrainer object and an EnemyTrainer object, each of which hold six pokemon. The Controller provides a series of methods that GUI calls to get the outcome of certain decisions made by the user. These decisions are what attack a user chooses, and whether the user chooses to switch out a pokemon. After being activated by the GUI, the Controller is able to get the outcomes of the entire battle and make all necessary changes to the Model by utilizing both the Simulator and the Calculator. In addition, the

Controller checks for win conditions and collaborates with the AIUtility to determine what move the Enemy trainer will choose.

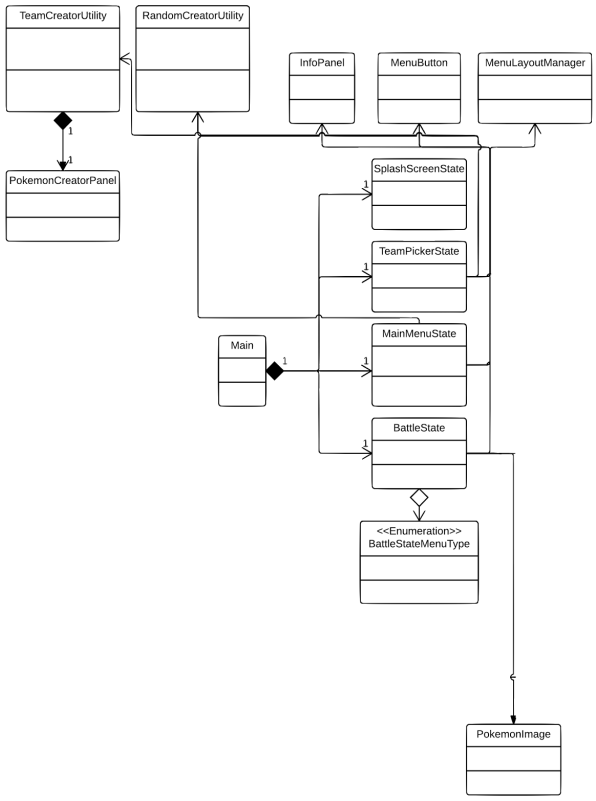
The way that the Controller and the Simulator communicate with the GUI is through an abstract class we called Events. There are six specific classes that implement this abstract class, each of which instruct the GUI to do something different. With so many Pokemon and so many options for moves and swapping, this approach allowed the Control-View relationship to be enormously versatile. Whenever a user input activates the BattleController object, all changes to the model are made, then an ArrayList of Events are given to the GUI to show step by step what happened.

The Events include the TextOutputEvent, which instructs the GUI to display a customized message held within the event; the PokemonSwitchEvent, which tells the GUI to remove the current pokemon from the screen and replace it with the Pokemon contained in the event; the UpdateHealthBarEvent, which tells the GUI to change the health of a Pokemon that the BattleSimulator damaged; the EnemyDefeatEvent, which tells the GUI that the user won; and the UserDefeatEvent, which tells the GUI that the user lost.



To implement the core parts of the game, we have created the Trainer, Pokemon, and Move classes. The Trainer class is able to store a list of Pokemon, the current Pokemon it has selected, and has a method to tell if any of its remaining Pokemon are living. This allows it work communicate with the battleSimulator and indicate if the user or enemy has lost (out of living Pokemon). The Pokemon class holds the moves of the Pokemon, the stats, and its national dex ID to have other methods get its image to be displayed. The Pokemon class also has a method, isAlive, which tells

the battleSimulator if a Pokemon's health goes down to zero. Lastly, the Moves class stores a couple of enums, indicating the type of move (same as Pokemon type), along with damage type. These enums, along with the damage and accuracy of the move, are used in the battle calculator to determine how much damage a move does. To make things easier, both userTrainer and enemyTrainer are children of the trainer class, and instances of each are stored in the PokeModel. The PokeModel is used to hold information of the user trainer, then is able to store an enemy trainer and load in a new enemy trainer with the trainer loader utility. Essentially, the PokeModel will always hold the information of the user, and each time it begins a battle with a new enemy, a new enemy trainer is loaded into the model.



The GUI for Pokemon Orange and Blue is based around the Slick2D StateBasedGame organization. This means that each mode in the game is considered to be a "state", and each state is responsible for handling input, updates and rendering while in the given state. This allows for the compartmentalization of the different states that the game needs to be in, and allows for drastically different layout and UX in different states.

Because the battles in Pokemon are exclusively menu based, one of the most important components to create for the GUI was a layout manager that could control the

highlighting, selecting, and rendering of buttons in a given menu. Thus the MenuLayoutManager was born. A MenuLayoutManager is created with a Rectangle that tells it over what area it's going to be laying out the buttons as well as the dimension of the matrix that it should store buttons in (i.e. a 2x3 will have 2 columns and 3 rows). The manager will then calculate the size and position of each button as necessary, so each button's position doesn't have to be worried about. The MenuLayoutManager is also special in that it uses generics, meaning that it can hold any class that extends MenuButton. This allowed for its reuse in many situations. An example of this is that the bars displaying the Pokemons' health was all done using a MenuLayoutManager, because InfoPanel extends MenuButton. Because all of the interactive graphical interface is done through the menus, all of the interaction is handled through the MenuLayoutManager.

The other significant hurdle with the GUI was handling the events and animations that came from working with the BattleControl. This was done by reading all of the returned events from the BattleControl into a LinkedBlockingQueue. This was then pulled from, one at a time with a delay being added every time an event was handled, and a new event being handled every time the delay reached zero. This meant that we could display text for a fixed amount of time to the user. To skip the text (which is done by hitting the spacebar while text is being displayed), the delay simply had to be set to zero.

The next issue was animating the Pokemon's images to allow for smooth switching as well as attacking and defending actions. This was done by creating a class (PokemonImage) that took in a rectangle in which it would draw the Pokemon. We needed the enemies' Pokemon to scale but the player's Pokemon needed to be cropped to not show over the menu. This was handled by creating an enum to decide whether to scale or crop the image, and this would in turn change whether we chopped off the bottom of an image that was too large, or merely resized it down. To actually animate the images, we kept track of a current offset, maximum offset, and the necessary constants to draw the image to the necessary offsets. The current offset was then updated every frame by including an update method that was called in the update method of the game loop.

The information about all of the Pokemon and the information about all of the moves that Pokemon can learn is stored in XML files that we found online. The PokemonLoaderUtility uses the jDom library to parse the XML files. The PokemonLoaderUtility makes it easy to extract information about Pokemon and Moves. Its public interface includes methods to get all Pokemon species names, get all moves names for a specified Pokemon, create a Pokemon object from a Pokemon species name, and creates a Move object from a specified move name. Also we created a Professors.xml which held all of the information about the professors and the Elite Four

trainers including their Pokemon, names, order number, and their speech text. TrainerLoaderUtility utilized this XML file to return a professor given which number in the order of professor the player is on (1-8 for the professors and 9-12 for the Elite Four). There also is a RandomCreatorUtility which can return a trainer with a specified number of Pokemon where the species of the Pokemon is random and all four moves of each Pokemon is random. There is also a method to return just a random Pokemon with four random moves.

The TeamCreatorUtility class holds a method that returns a Pokemon object from user input. The method creates and shows a custom jPanel using Java Swing API. The panel, PokemonCreatorPanel, has a drop down list that has all of the available Pokemon to choose from. When a Pokemon is chosen, another panel appears which has four drop down lists where each one is populated with all the possible moves that the chosen Pokemon can learn. Once the user chooses at least one move, then clicking the Ok button will create a Pokemon object with the specified species and moves. Control is then returned to the method in TeamCreatorUtility where it extracts the information from the panel and produces a Pokemon object using PokemonLoaderUtility.