LINGI1131 - Pokemoz

Rémi Chauvenne
Jerry Wei

May 2015

1 Introduction

The objective of this project was to create a simulation of Pokemoz trainers, Pokemoz, fights against wild Pokemoz, and other Pokemoz trainers. The goal of the player is to make their way to the home of the Pokemoz Prof, navigating roads and tall grass and battling any wild Pokemoz or trainers that they encounter, while evolving and keeping their Pokemoz alive throughout the game. We started the project by first thinking out the basic structure of the program and how to organize our code in different files. Then we used QTK to create and load graphics along with our map. Secondly, we used the port object abstraction to create port objects as playable trainers and Pokemoz. In addition, we utilized message-passing concurrency to communicate between port objects and trainers. Finally, we went through again to make sure no race conditions or violations of game rules existed.

2 Diagrams

2.1 Components Diagrams

The getState message can be sent both ways by both Trainer port objects to each other, whereas die can only be invoked on the PC's Trainer. This is because when P1's Trainer has 0 remaining HP, the game is over and the window will display this.

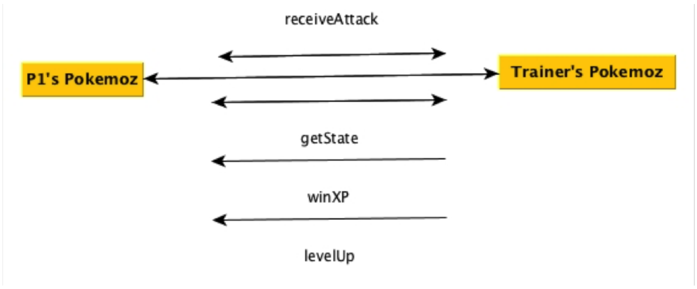

Figure 1: Pokemoz component diagram

The getState and receiveAttack message can be sent both ways by both Pokemoz port objects to each other, whereas winXP and levelUP can only be invoked on our P1's pokemoz. This is because in these cases only the playable Trainer's pokemoz matter.

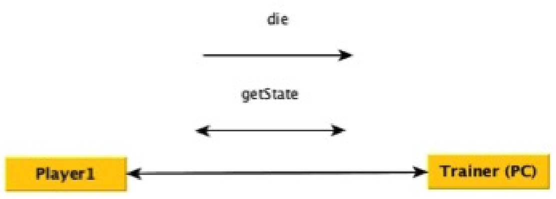

Figure 2: Trainers component diagram

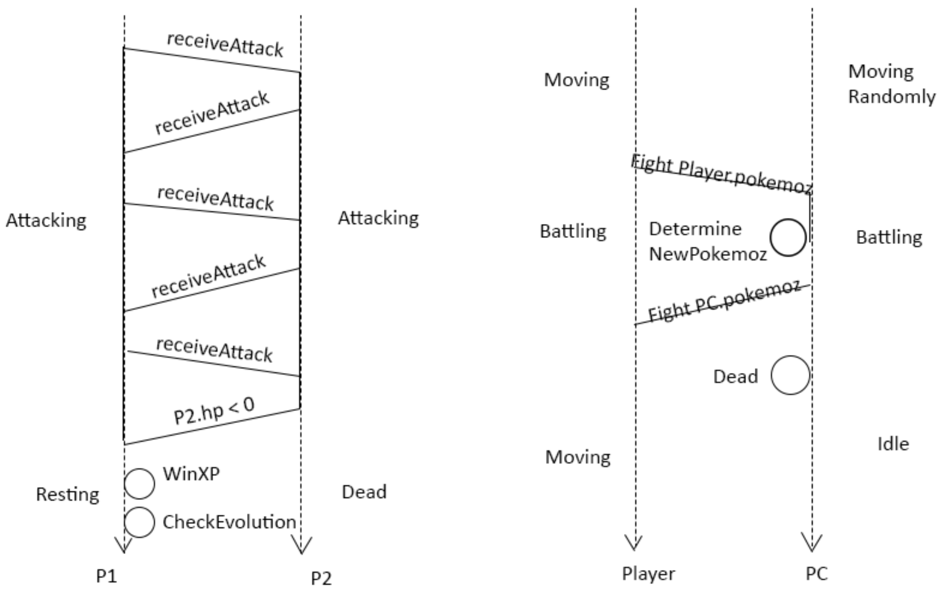

Figure 3: States diagram

2.2 State Diagrams

3 Data Structures

3.1 Port Objects

Port Objects are ports (which in themselves are streams) with an internal sequential state. We used port objects as the data structures to represent objects in our program (Trainers and Pokemoz), as they serve the same primary functions as an object would in Object-Oriented Programming. The internal state in our port objects is written in a record (explained below), which corresponds to attributes or class variables in OOP. This is coupled with a port object abstraction function which is a simple function perpetuating a stream. Finally, there are "messages" we send via the Send function to each port object to invoke a particular function necessary of the port object (e.g. maxHealth, levelUp, etc).

3.2 Records

We utilized a record to define the state of our state objects. Records are advantageous in this situation because they can effectively store more than one piece of information in a port object at once (i.e. Health points, Experience points, etc). One interesting matter to note is as we are programming in the declarative paradigm, these states are immutable and therefore cannot be changed. Thus, the solution to this is to re-assign each variable in the state to a new record, and assign/return the new state to replace the old state. With this process, we can edit a variable in the state (add health points, evolve Pokemoz level) while still conforming to our programming paradigm.

4 Conclusion

In conclusion, we discovered that there were disparities between what we thought were the more difficult parts of the project and the actual complexity of the code once we wrote it. The majority of the work for this project revolved around port objects and employing message-passing concurrency. While we originally thought loading the map and using functors was going to be a difficult task, we instead found that proceeding to the logical next step after building our map using QTK was a more arduous task. In response to this, we collaborated our ideas to write different parts of the project. Ultimately, we finished the project by incorporating components of the game together.