John Casillas
Corbin Stripling
Tristan Santiago
February 18, 2018
CS162-400
Group Project Reflection

We began this project by closely reviewing the specifications listed in the assignment guidelines. Right away, we knew we would be able to re-use code from Project 1, so we immediately made it a point to revisit that assignment before starting on this project. We took this as an opportunity to compare code and determine whose code we thought would be best-suited for this task. In the end, we ended up picking up various bits from everyone's code to form what we thought to be the best possible version.

The process of implementing code from Project 1 really guided our design process, as we really just proceeded by tailoring the old to fit with the new. Since we knew we had two types of insects for this game, we thought it would be best to create a base critter class that would define the characteristics both the ants and doodlebugs would use. For example, we knew both ants and doodlebugs would need a function to move, another function to determine whether or not the ant or doodlebug has moved, and a function to handle death, so we knew that these functions would be fit perfectly in the base class. Our base class also included a function to determine bug type, which we knew would be used later on in the program. Finally, we included both a breed and starve function in the base class. Realizing that ants don't starve, we still thought this would be the best place to declare those functions, because doodlebugs could starve and breed, so it needed to inherit both, whereas ants only needed to inherit breeding from the base class. This method felt like the best approach because we were saving ourselves the trouble of having to define these functions multiple times by placing them in the base class, as we could simply define the specifics of those inherited functions later on in the respective class.

Once we completed the classes, building the program was rather straightforward. We simply followed the assignment instructions, adding one piece at a time and testing it before moving forward. Our first objective was to successfully create a 20x20 array for the game board, place a single ant, and have that ant move randomly, which we accomplished with relative ease. Once we got the ant's movement working, we then set each element of that array to null (empty) and began working on randomly filling the array with 100 ants in random locations. This proved quite tricky at first, so we ended up having to creatively solve the problem by working backwards. Initially, it was quite easy to fill the entire array with ants, but obviously, 400 was way more ants than we needed. We then tried modifying the code in such a way that it would loop through the entire array and place ants randomly, but ran into an issue where sometimes all of the ants wouldn't be placed. While they were being placed randomly, the program wasn't always placing 100 of them. We thought this was likely due to the fact that sometimes the random rolls were hitting some of the same elements in the array twice and we couldn't really create any logic to properly handle those cases, so once again, we took a step backwards.

The next idea we came up with was simply creating distinct variables that would store the number of ants and doodlebugs each time they were added. We used those variables in a while loop and used the random roll function for both columns and rows to generate a spot in the array. Once that spot was generated, we ran a check to see if the spot was empty. If the spot is empty, we placed the critter there and incremented the respective critter count. This loop continued until the appropriate number of critters had been placed, removing the need for us to actually introduce additional checks, which we found was necessary with if/else statements, causing the aforementioned issues.

Here is an example of our code in action for adding ants:

```
…
// Randomly place 100 ants.
int antCount = 0;

// Initialize a while-loop that continues to execute until 100 ants have been successfully placed.
   while (antCount < 100)
   {

      int i = rand() % 20;            // Generate a random position from 20 columns.
      int j = rand() % 20;            // And also generate a random position from 20 rows.
      // To determine our random placement for the ant on the game board.
      // If the spot is NULL, then it is empty, since we initialized the entire array to NULL
      // beforehand.

      if (grid[i][j] == NULL)         // Then only place an ant in an empty spot.
      {
        grid[i][j] = new Ant(i, j);   // Create an ant at the specified random location.
         antCount++;                  // Increment the ant count, so that we make sure
                                      // we're placing 100 ants.
      }
   }  // End while
…
```

We applied the same logic to doodlebugs and found ourselves with a program that successfully placed critters in random locations. From there, we needed to add our logic for doodlebug movement and behavior, since ant was already completed in an earlier stage. Once we completed that, we simply needed to prompt the user for how many steps to take and execute movement and behavior for each one of those steps. Once the steps were completed, we implemented code to prompt the user to run the simulation again.

This week's workload proved quite challenging for all of us this week. We did well to get started on this project early, but eventually the limited time frame in addition to the extra work due this week prevented us from spending as much time working together as we would have preferred. We also made a critical mistake of not using valgrind on our program until we completed it. Perhaps we were too caught up in testing the other

aspects of the program that we forgot to check for memory leaks. The result was a finished product that compiles and executes as we expected, but still ends up leaking memory according to valgrind after the complete execution of the program, no matter how many steps we take or how many times the simulation is run. We worked diligently and to the best of our abilities to trace these memory leaks right up until the deadline, but, much to our dismay, we were unsuccessful in being able to locate the source. At the point of realizing that our program was leaking memory, it was far too late to re-write the entire program in hopes of detecting the leaks earlier, or avoiding them altogether.

Overall, we were very satisfied with the final product, despite the last minute detection of memory leaks. We were as fascinated with the execution of the program from holding the ENTER key, as we were perplexed with having memory leaks in a program that appears to work appropriately on the surface. While we regret not being able to successfully resolve the memory leaks, we remain proud of the work we were able to accomplish with this group project.