Reflection
Assignment 1
Tyler Forrester

I. Design Document

The objective of this assignment was to improve our understanding of polymorphism and our use of dynamic arrays. The assignment was to create a 20 by 20 grid that contains Ants and Doodlebugs and Empty Spaces. The Ants and Doodlebugs have some abilities that resemble real life. They can eat, move, breed and die. Although Ants and Doodlebugs have same method of movement, they differed in their ability to eat, breed and die. Ants were able to only eat on "Empty Spaces" while Doodlebugs only ate if moved into the same space as an Ant. This reset the time to death counter for doodlebugs to 0. Ants never got to reset the time to death calculator, but were blessed with longer lives, 10 time increments compared to 3 for Doodlebugs. The final advantage the Ants had over the Doodlebugs is that they breed every 3 rounds where a Doodlebug breed every 8.

Since both Doodlebugs and Ants inhabit the same world and have the same basic functionality, it made sense to create a base class that kept track of attributes that they both shared. This meant the "Critter" class needed to contain a move, eat, death, and breed functions along with get functions for both the time to die counter and the time to breed counter. The move function returned a seeded random number between 1 and 4. This was translated into the main function as a direction, north, south, east or west. The creature was allowed to move if the space was currently unoccupied or it was Doodlebug moving into an Ants square at which point in time the ant would be killed. When the Doodlebug moved into the Ant's square, the eat function was called and the D-bugs time to death counter was reset. These were stated requirements, so the code was implement to meet them.

The death function was to be called if an Ant had lived 10 turns or a D-bug had lived 3 turns. Since both D-bug and Ant had to different requirement to die, I chose to turn the death function into virtual Boolean function, which would return true when called. This seemed to fit the idea of encapsulation the best. It allows both Ant and D-bug to have both different requirements for death, but recognized that both animals were going to die, so it was clearly a shared trait. The virtual function let me override the base death function for each creature, which allowed me to set the Ant timer to 10 moves and the D-bug time to 3 – moves or an eaten ant. Both timers were dictated in the eat function, which was perhaps better named the hunger function as the only time it reset was when a doodlebug ate an Ant.

The breed function had similar structure to the death function, so I also turned it into a virtual function. Both creatures bred, but both creatures had a different time frame for doing -- although they could use the same timer. Since they could use the same timer, I chose to keep the timer in the Critter class and increment each time it was checked by either the Ant or the Doodlebug Class. This encapsulated critter and insect behavior and would allow more insects to be added to the Critter board.

I also added a name to the critter class for identification of Doodlebugs and Ants on the board and also to have some physical representations of the different types of objects represented by critters. I chose an "A" for Ant, "D" for Doodlebug and a "." for uninhabited spaces. This felts like it gave a nice visual effect to the creatures moving through their lifecycles on the screen. It also gave me an easy test for what inhabited board spaces, since I had the character representations, it seemed easiest to take those characters as logical representations as the "A", "D", and "." I'm sure that for future enhancements dynamic casting would be preferred however I did find an academic paper that argued that an "ID" approach was actually more efficient ("Fast dynamic casting - Bjarne Stroustrup"), so there still seems to be debate in the academic community.

This brings use to my main class. I chose to randomly assign 5 D-bugs and 100 Ants to spaces. This was per my TA's request. I used a 20 x 20 Critter pointer array for my board. The pointer array allowed me to use virtual functions which improved my encapsulation. It added some steps to my main as I then had to initialize the board to new critters and then populate with new D-bugs and Ants. After the population, I chose to pause my program as an Intro to the differential equation. The board prints out with a heading announcing the representation and asks the user to enter "c" to continue. I use an Input Validation class I wrote for Assignment 1 to validate the char. I liked the aesthetic. A

After it continues, the program proceeds to count populated squares to figure out if it should continue running the function. The function should stop after every square is blank or every square is an Ant or a Debug. It also resets the counter that determined if an object was moved in this time frame. The array is processed from the top right to the bottom left. If an object has been moved in the current turn, it will not be moved again. However, since the processing happens sequentially occasionally an object will be unable to move forward on a turn, because its movement was blocked by another object. This object may yet to be processed, so can be moved forward another square, creating a gap between two objects moving in the same direction. It only occurs moving east and south. This was a design decision that balanced the need for efficiency with the need for accuracy. By only allowing an object to move once per round, it accomplished the goal of not allowing the objects to live in different continuums while still keeping the coding to (sort of) manageable size.

After the board has been counted and flags clear, we move into the movement section of main. The array cycles through all of it elements stopping only when it finds a square that is not a blank and has not been moved this turn. This meets the encapsulation goals better as this will be extendable to future bug classes. Once a bug is found in the array, its breed function is called. It was requested in the documentation that it be called before he move function. This populates a breed variable which is either true or false. Then the move function is called, it returns a random number between 1 and 4. These sets of a list of if statements that check for the number and if the movement will remain on the array. If the insect will not remain on the remain an else statement is called to iterate to his time of death. If the movement remains on the array, then the j or I change function is called. Both functions have the same logic, but one allows movement east and west and the other north and south. In a future state, I should find some time to find the trick to combine these logic statements. However under time constraints they are going to stay as separate functions. I and J change are the meat of the controllers logic. I broke the class into four sections, if the creature is an Ant and the space is unoccupied, call the eat function to mark the passage of time, then check and see if the Ant has died. If hasn't died move the ant. If the Ant has breed, populate the space left as a new Ant. Otherwise populate it as a new Critter. Case 2: Repeat the same logic for Doodlebug, except creating a Doodlebug as offspring. Case 3: Doodlebug moves onto an Ant. This calls the eat function as passes true Boolean to it, which will trigger the eat function to reset the D-bugs time till death counter to Zero. Repeat the same logic for the other parts of the function. Case 4 – Insect can't move, because the space is blocked. Call the time till death counter and check to see if Insect has died.

When these statements finish the board is then printed out and is paused for 1 second using some code I found on the Ubuntu Forums. It keeps checking the time until the number of seconds have passed that I entered. The statement then loops until either all creatures are killed or there are no empty squares.

II. Testing Plan

My testing plan was to find the right balance between thoroughness and efficiency. I chose to test my project as I built. Only starting the next section once the prior section had been adequately tested. The basic plan was to build a display and then test individual pieces of the code, like moving in 4 directions before finally combining all the pieces. It felt effective as the tests layered, I had confidence that my program would work as expected. The test results are below:

III. Test Results

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Board position with Ant object displays "A"	none	Main()	" A " in 3 3 otherwise " . "	" A " in 3 3 otherwise " . "
"A" moves in direction indicated 1 – West 2- South 3 – East 4 - North		Main Move()	Moves "A" to 3 2, 4 3, 3 4, 2 3	Moves "A" to 3 2, 4 3, 3 4, 2 3
"A" moves in direction does not move off board	A is in [19][19] Moves South and East	Main()	A stays in place	A stayed in place
"A" moves in direction does not move off board	A is in [0][0] Moves North and West	Main()	A stays in place	A stayed in place
"A" moves into a "D" and stops.	A is in [0][1] Moves West	Main()	A stays put	A stayed put
"D" moves into a "A" and eats A	D is in [0][1] Moves West	Main()	D eats A	D ate A
If breed equals "true" "D" replicates	D is in [0][1] Moves West	Breed()	D leaves a D behind	D left a D behind
If eat count equals "3" "D" disappears	Test by moving 3 moves in time	Eat() Death()	D is removed for array	D is removed for array
If eat count equals "10" "A" disappears	Test by moving 10 moves in time	Eat() Death()	A is removed from array	A is removed from array
"A" replicates after 3 moves and D replicates after 8 moves	Test by putting A directly infront of D. As the array is process A moves forward and D follows. When A replicates D eats A which allows D to keep living and replicate after 8 moves	Eat() Death()	D replicates several times. A replicates and is eaten by D	D replicates several times. A replicates and is eaten by D

Screen Paused	None	Wait()	Pauses screen briefly between time rounds	Paused screen briefly between time rounds
If eat count equals “10” “and A hasn’t moved “A” disappears	Test by taking 10 time steps moves in time	Eat() Death()	A is removed from array	A is removed from array

IV. Reflection

The program was long trying to decipher the requirements was difficult and per the last several assignments the requirements changed midway through or more requirements were added. The only way to resolve these problems is to work ungodly hours. As to the actual coding, I tried to be efficient in my use of time and not become too attached to any one style before the requirements were more fully fleshed out. The only real major issue I ran into was I had written my program originally an Array of Critters. This worked fine with by passing name into the Critter Class for all items, however I couldn’t encapsulate the breed or death functions. This required a refactor to move their functionality back to their respective classes. And ultimately allowed forced me to rewrite the program with dynamic arrays. This seemed to much improve the encapsulation of the classes.