Group 8:

CS 162 Fall 2018

November 4, 2018

**Group Project: Predator-Prey Game**

Design & Reflection

**Program Design: The Problem**

Ants and doodlebugs live in a 20 \* 20 grid of cells.  Only one critter may occupy a cell at a time. The grid is enclosed and no critter may move off the grid.  Time is simulated in steps. Each critter performs some actions every time step.

**Game Flow:**

The world will start with 5 doodlebugs and 100 ants.  They will be randomly placed on the grid and the user will be prompted to enter the number of time steps to run.

For each time step:

1. Bugs will move
   1. The doodlebugs will move before the ants in each time step
2. When breeding, eating, and starving are resolved, display the resulting grid
3. Draw the world using ASCII characters of “O” for an ant, “X” for a doodlebug and blank space for an empty space (the characters should be arranged to look like a grid)
4. When the time steps entered by the user has been reached, ask them to enter another number and start to run the simulation again or to exit the program.
   1. You must maintain the state of the current grid while creating the next display.

**Program Design: Modularizing & Structure of Program**

During our first group meeting, we discussed the program design. We decided on having the class structure below:

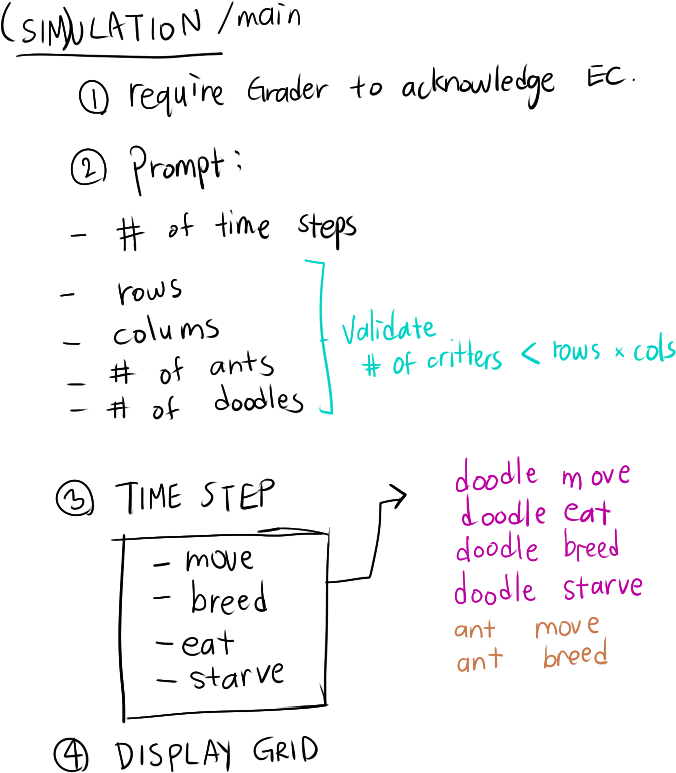
* Critter – The Base Class to contain all functions and variables common to both Ant/DoodleBug
  + Getters/setters
  + Base Constructor
  + Variables – row, col, timeSteps, state
* Ant – Derived Class to contain its own dedicated move/breed functions
  + Custom move to return a direction based on ant’s surroundings
  + Custom breed to return a direction based on ant’s surroundings
* DoodleBug – Derived Class to contain its own dedicated move/breed functions
  + Custom move to return a direction based on DoodleBug’s surroundings
  + Custom breed to return a direction based on DoodleBug’s surroundings

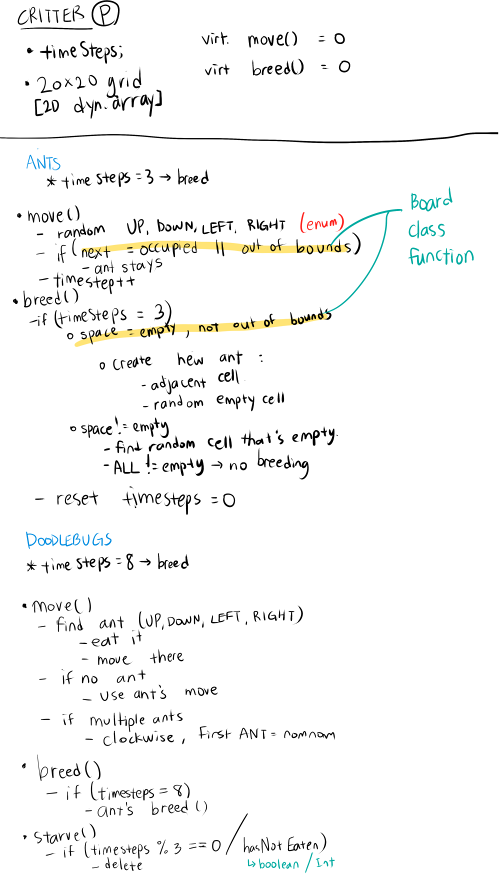
Those three were fairly simple to decide on, mainly because they were required in the specs. The next question was how we would handle the grid and simulation itself. We decided that it would be fine to combine the simulation/grid together into a single class, but we would want it separate from Main. So that way all main had to do was launch the program. Eventually that ended up being our game class below.

* Game – Creates and destroys the grid of critters (Ants and DoodleBugs), and runs order of simulation by processing moves, breeds, and starves.
  + createGrid – Make the grid of critters
  + deleteGrid – destroys the grid of critters
  + runSim – actually process the simulation based on the number of steps
  + move/breed/starve – manipulate critter objects based on critter functions
  + dirSelect – determine the surroundings of the critter

The next discussion point was the Menu/Input Validation. We decided that since we each had our own working menu/input validation function, whoever eventually implemented the menu/input validation should just use their own relevant functions and classes. That was because it would be the most familiar to the coder, and no one particular menu had a major advantage.

Below are the notes from our first design meeting:





**Program Implementation: Process**

After our first meeting, we decided it would be best for everyone to start working on the implementation separately so we could get a better feel for the project. Certain specifics such as how the critter should know it’s surrounding or how to move critters without double counting were tough for us to nail down during our first design meeting. We thought if we each tried coding a little bit, then we would feel more comfortable with the project and concepts.

After a few days we had each tried the overall skeleton of the project, and were ready to begin the final implementation.

Greg’s work so far was the easiest to build off of, so we had all agreed to work off the foundation that Greg had laid. We used GitHub to make the necessary modifications, and different team members would approve pull requests to make sure there was always a code review process. Github turned out to be essential for our collaboration to easily stay on track.

**Program Implementation: Specific Decisions**

During the implementation, our team had to make a few specific decisions based on how we thought we could satisfy the specifications.

* *Current Grid State* – Originally, we thought the current grid state requirement meant that during each time step we had to keep a copy of the grid before making any modifications. Later after the professor’s piazza post we realized that it actually meant that the play again option should continue where the first simulation left off. Since we had already coded the play again option, we took the Professor’s available option to leave our code as is. We did decide though to streamline our relevant move code to take advantage of the fact that only a single grid state is required. Having less grid states to track made our conditional statements much cleaner and shorter.
* *Which functionality to include in Critter vs Game* – There was a consistent question of what information should stay in the game class, and what access the critter would have. The critter is ultimately the object that would do the move, but we felt, based on OOP principles, that the game was the class which should know the surroundings of the critter. So we decided that the game would send information to the ant/doodlebug move function, which would allow the critter to make the move decision. After the move decision was made, the game would receive the decision and update the grid.
* *Location of Starve Function* – Based on the rubrics/specs it confused us where exactly the Doodlebug’s starve function should take place. The specs said that the only critter functions should be the ones in common between doodlebug and ant, but the rubric said that starve should be overridden. For starve to be overridden, that would imply though that starve was a function in critter even if it was only used by doodlebug. We decided to follow the spec, and keep critter functions only to the ones which are in common between doodlebug and ant. So we then handled the starve logic within the game class. This implementation choice also allowed us to use critter pointers freely, and without having to worry about any dynamic casting to call the starve function.
* *Placing vs Swapping for Moving* – During the implementation we tried/debated two different ways for handling movement. Because the final grid design we used had nullptrs mixed with critter pointers, we had the option of moving/placing critters freely on empty spaces. However, Daniel’s original concept had a grid filled with base critter pointers mixed with specific ant/doodlebug pointers, so it made more sense from a memory perspective to move critters by swapping them with the pointer object in their new space. Ultimately the internal logic for the final version made more sense to “place” critters directly in their specific spots. It also simplified the code since there were no swapping requirements. The team did have to troubleshoot and resolve a memory loss issue though, since the team realized that swapping had preserved memory in the event that a critter was placed onto the same spot it began. (The case when a critter did not move).