# Assignment 2 – Inheritance and 2D Arrays

**Goals-**

Develop classes from given requirements

Implement those classes using object-oriented programming techniques

You will design, implement, and test a cellular automata program. The program will simulate a simple predator-prey model. You will have a 2D array populated by Ants, Doodlebugs and empty spaces. The Critters (Ants and Doodlebugs) will move around, periodically creating offspring. If a Doodlebug moves into a square containing an Ant it will eat the Ant. If a Doodlebug does not eat for 3 steps it will die. Ants eat whatever is in the space and will die after 10 time steps.

You must create a design document. The class hierarchy is simple. You will have a Critter base class. Ant and Doodlebug will be subclasses. What do they have in common? Those elements will go in the Critter parent class.

**Movement:** The 2D array will have a data type of pointer to Critter. That allows either type of Critter to be in any element of the array. Movement is random. The move function just returns one of the 4 possible directions the Critter can move, determined at random by the object. The main function will hold the array. For each existing creature it will call its move function and then move the object in that direction. If it cannot move in that direction it remains in the same element. The Critter will have no knowledge of its location; it just wants to move.

**Breeding**: It must know the number of turns since it last bred. Before a Critter moves the main program must call its breed function. If the counter has reached 3 for an Ant or 8 for a Doodlebug then when the Critter moves a new object must also be left in the original position. If the Critter cannot move then it cannot breed. Nothing changes in the original Critter! When a new Critter is bred, the breed count resets to 0 and time moves on. When the breed function is called and the counter is not at the limit, the function will simply increment it by one.

**Eating**: When a Doodlebug moves onto an Ant, it eats the Ant. The pointer to the Ant object is simply freed in the main program. The program also must call the eat function. This function simply resets the counter if the Doodlebug ate an Ant. If the grid element holds and Ant or a Doodlebug that did not eat then the function increments the counter for the Critter. You can use a Boolean parameter to indicate if the Doodlebug ate an Ant. When eat is called for an Ant then it will ignore the parameter.

**Death**: After the move and eat functions have been call, the main program must call the die function. For either Critter, if its counter exceeds the time to live counter (i.e. 10 for Ants and 3 for Doodlebugs) then the Critter dies and is removed from the grid. This will be a Boolean function.

**The program**: What will the main program look like? You will need a nested loop to run through the array. For each occupied element you must call the move, breed, eat, and die functions

First create a draft of your design. You must have the class hierarchy with the behavior of the functions specified for each class and subclass. The functions are created with different limits for their counters. Eat for a Doodlebug may also reset the time to live counter.

IMPORTANT NOTE: All behavior for a class must be contained within that class. If you are doing some activity for a class outside the class then you are not enforcing encapsulation.

The main program is simply doing the bookkeeping of which array elements hold which type of Critter. Then it calls the functions for the Critter object in each element.

After you develop your draft you can look at the last page of this document. It summarizes these requirements. It is not a complete design! It may not even mean much to you if you have not thought about the program for yourself.

The main program will manage the array. Use a 20x20 static array. Display the grid with Ants, Doodlebugs and empty spaces at each time step. Pause the display at each step so the changes do not fly by. All the counters will start at 0. You will start with 5 Doodlebugs and 100 Ants. For debugging you should have each Critter start in the same location. You may adjust the starting number of each Critter in case the grid fills up, or one or both Critters consistently die out. Remember that if you do that you would also need to add or remove Critters from the starting configuration.

**You will submit**- design document, test plan and results (these can be in your reflections but mark them clearly), and the source and header files- Critter, Ant, Doodlebug, main, and your makefile.. Submit all in a zip file.

Grading:

- programming style and documentation (10%)
- create the Critter parent class (10%)
- create the Ant subclass (10%)
- create the Doodlebug subclass (10%)
- correctly move Critters (10%)
- correctly breed new Critters (10%)
- correctly remove dead Critters (10%)
- correctly have Doodlebugs move and eat Ants (10%)
- display the grid of Ants, Doodlebugs and empty spaces at each time step (5%)
- reflections document to include the design description, test plan, test results, and comments about how you resolved problems during the assignment (15%)

Critter
timeToBreedCounter
timeToDeathCounter

eat()
move()
breed()
died()

| Critter | Time ticks to death | Time ticks til breed |
|---|---|---|
| Ant | 10 | 3 |
| Doodlebug | 3 | 8 |

The eat() function for the Ant class will just increment the timeToDeathCounter. The eat() function for the Doodlebug will increment the timeToDeathCounter unless it moved onto an Ant. If it did then the function will reset the timeToDeathCounter to 0. You can add a parameter to tell the Doodlebug object what to do. The Ant class would ignore the parameter.

The move() function simply returns one of 4 four directions determined at random by the object. It will increment the timeToBreedCounter,

The breed() function returns true of false. If it returns true then the main function must create a new object of the same class to put in the original element when the object is moved.

The died() function also returns true or false. If it returns true then the object is removed. That element in the array becomes empty.

Do your design first! Develop a class hierarchy and complete the specification of the functions.

What order will the main program call each of the functions for each object in the array? Remember that the move() function only returns a direction. The object is only moved when the main program moves it in the array.

There is an important design decision required. If you process the array from the upper left corner, doing each element in that row and moving down to the next, what do you do if you move the Critter object down or to the right? As you move through the array you will process that object again. And possibly again. And possibly again. Each object should be processed only one time each cycle or generation.