

Group 25

Derth Adams

Yuxing Deng

Manda Jensen (Phadke)

Kayla Smith

Phillip Wellheuser

Workload Plan:

We all uploaded our initial design thoughts by April 27th and had a voice conference for about an hour and a half to discuss our game plan. We decided to generally go with the plan that Derth laid out, as it was the most in-depth and thought out. We determined that Kayla would be in charge of documentation, reflection, and testing. Derth was in charge of the Grid class, Manda and Phillip were in charge of Critter, Ant and Doodlebug, and Yuxing was in charge of main and Game. The initial hpp files will be uploaded to the files section by EOD Tuesday, April 30.

The main issue in the workload division was deciding how the Critter classes should interface with the Grid class. This was hashed out through group discussion in our Slack channel. We discussed coding through github or some similar platform, but determined that not enough people in the group had the time or experience necessary to successfully navigate that along with the group assignment. From there, we decided to code our individual portions on whatever platform we chose, then upload the files to a folder in our Canvas group. Updated iterations of code would be moved to an old source code folder if they were phased out. Once all the code was updated, Kayla went in, created the makefile and tested the program.

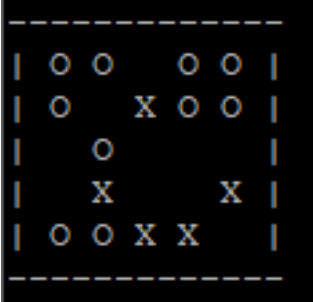
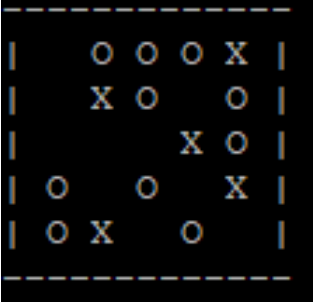
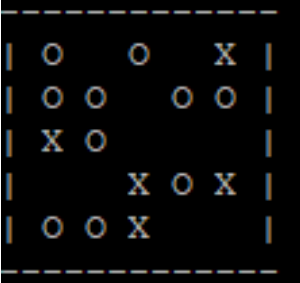
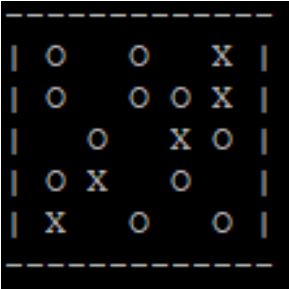
Project Design Plan:

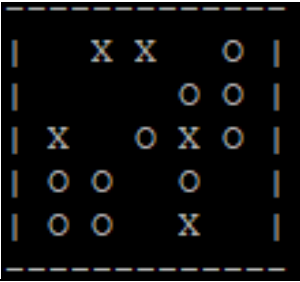
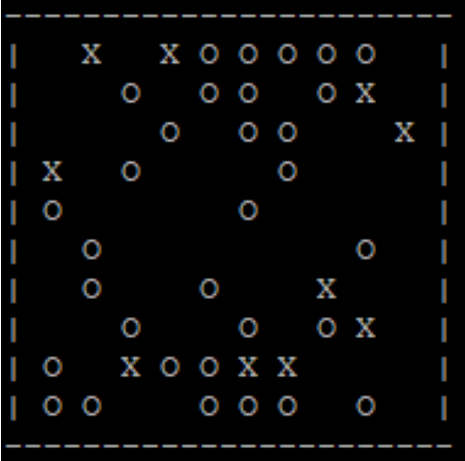
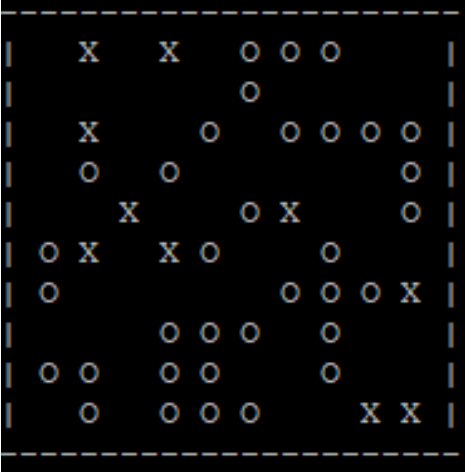
Derth's design plan is included at the end of this file (pages 12-17). This is the plan that we most closely emulated in completing the assignment.

Testing Plan:

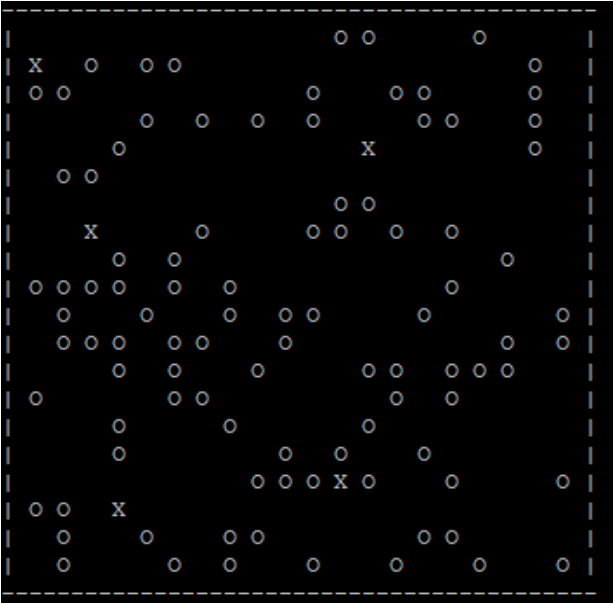
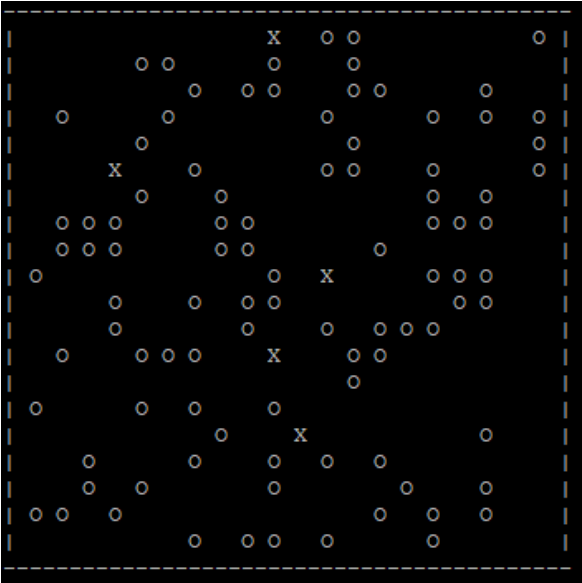
Test	Reasoning	Expected Result	Exact input	Actual Result
Input a very large (10,000+) number of steps. Since max number of steps per input is set to 1,000, this will be done in 1,000-step increments over 10 prompts.	Make sure that the program will continue to run given a high volume of steps.	Program will run as usual and likely run out of critters at some point.	Game set to 80x80 and with 30 Doodlebugs, 500 Ants. Ran 1,000 steps, then when re-prompted continued 10 more times at 1,000 steps each prompt. Then continued for another 100, then 900 steps	Program ran as normal, with described cyclical nature of Doodlebug and Ant populations observed. Correct number of steps run at each input prompt. No segmentation faults or other crashes observed.

			to verify the value for number of steps was being changed.	
EC: Try to set very small grid with more ants and doodlebugs than can fit on board	Make sure the program can recognize and resolve this case.	Re-prompt for new values	Test 1: 10x10 board, 5 Doodlebugs, 100 Ants Test 2: 10x10 board, 200 Doodlebugs	Test 1: When entry for 100 Ants was entered, the program prompted for an integer between 1 and 95 Test 2: When entry for 200 Doodlebugs was entered, program re-prompted for an integer between 1 and 99.
Do multiple runs (several continuations)	Make sure the program does not reset the board when more steps are entered	Program will continue with the same board for the specified number of steps	Set 10x10 board, 15 Doodlebugs, 70 Ants, 5 steps to start, then continued for 5 more steps 10 more times for a total of 55 steps	Program ran as expected and continued with same board at each prompt for continuation
Test all user input with invalid entries (negative numbers, strings, floats, characters, zero)	Ensure that only valid integer inputs are accepted	Program will reject all invalid entries.	(EC) Tests for entering rows of grid: 0, "abcde", 81, -80, no input, 'z', 9.7, '+'	All invalid inputs rejected by program and re-prompted.
			(EC) Tests for entering columns of grid: 0, "wxyz", 81, -80, no input, 'a', 11.11, '='	All invalid inputs rejected by program and re-prompted.
			(EC) Tests for entering number of Doodlebugs: 0, "asdf", 'q', -5, no input, "1/2", 6.7, '['	All invalid inputs rejected by program. Note: Doodlebug max inputs tested with board size above, so here we only checked lower bounds and non-integer values.
			(EC) Tests for entering number of Ants: 0, "lkjh", 'p', no input, "9/8", 7.8, '&'	All invalid inputs rejected by program. Note: Ant max inputs tested with board sizing above, so here we only checked lower bounds and non-integer values.

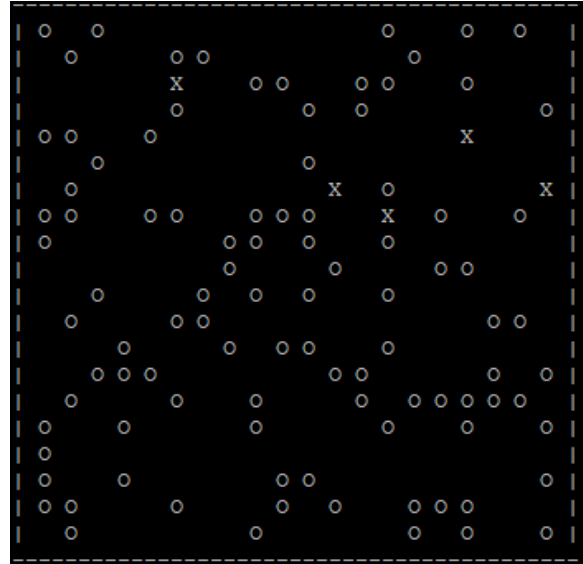
Run several times (~5) with a given board size and observe placements of ants and doodlebugs. Do this for several board sizes.	Make sure critters are not placed in same locations every time.	Critter starting placements will be different for each run.	Test 1: 5x5 grid, 5 doodlebugs, 15 ants, 1 step
			<p>Result #1:</p>  <p>Result #2:</p>  <p>Result #3:</p>  <p>Result #4:</p> 

			<div>Result #5:</div> <div></div>
			Test 2: 10x10 grid, 10 doodlebugs, 40 ants, 1 step.
			<div>Result #1:</div> <div></div> <div>Result #2:</div> <div></div>

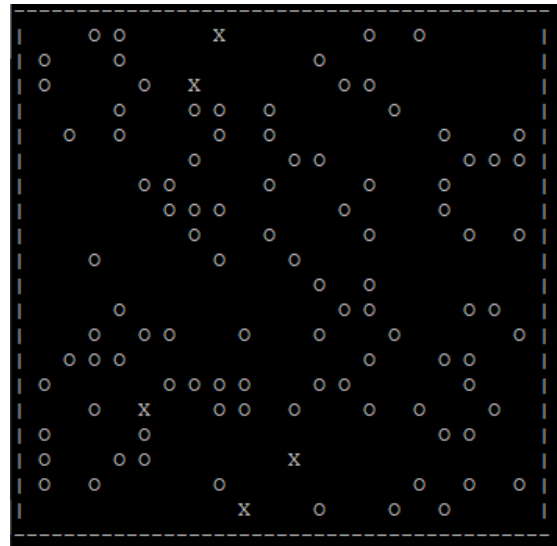
			<div>Result #3:</div>
			<div>Result #4:</div>
			<div>Result #5:</div>
			Test 3: 20x20 board, 5 Doodlebugs, 100 Ants (base specs), 1 step

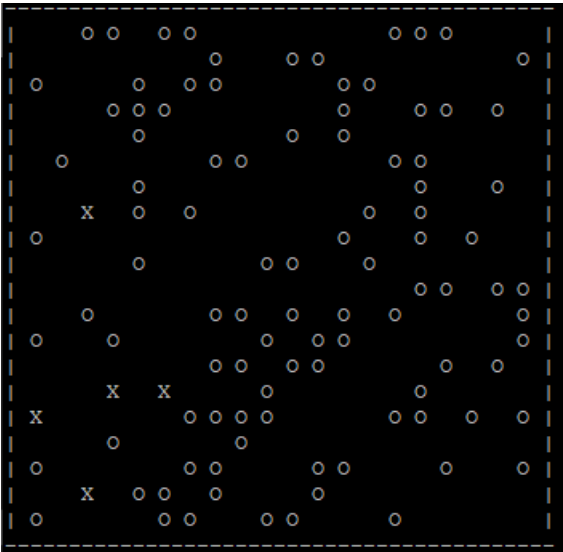
			<div>Result #1:</div>  <div>Result #2:</div> 
--	--	--	--

Result #3:



Result #4:



			<p>Result #5:</p> 	
			Results analysis: Doodlebug and Ant placements are sufficiently random for the scope of the project. No repeated setups were observed.	
Run several times and observe step-by-step the movements of the ants.	Ensure movements are consistent with assignment specs – each turn ants move after doodlebugs so they cannot “cheat death.” Ants move in a random direction, but cannot move into occupied spots or off the grid. If no “legal” options are available the	Ant movements are consistent with specs.	5x5 grid, 1 Doodlebug, 1 Ant, steps ran in groups of 3	Doodlebug died after 3 steps and Ant spawned a second Ant. From there, Ant multiplied as expected, breeding every 3 steps (or more if a situation arose where an Ant could not breed on its third turn). Movements are one step in a given direction, if they occur, as expected.

	ant does not move.			
Run several times and observe step-by-step the movements of the doodlebugs.	Ensure movements are consistent with assignment specs – each turn doodlebugs move one square to eat an ant if possible. If more than one option is available choose one ant at random to eat. Otherwise move randomly. All doodlebugs move before ants, so ants cannot move out of the way. Doodlebugs do not try to step into a space where another doodlebug sits or off the board. If no “legal” options are available the doodlebug does not move.	Doodlebug movements are consistent with specs.	5x5 grid, 1 Doodlebug, 24 Ants run in 3 step increments.	Doodlebug moves 1 space to consume an ant each step for first 7 steps, then spawns another Doodlebug on step 8. No cases of Doodlebugs eating Doodlebugs. Consistent with specs.
Run iteration with	Make sure that no new ants spawn	No new ants will spawn, no errors	Due to design, this exact scenario was not	Once Doodlebug died, Ants overtook grid entirely. Upon

completely full board of ants.	out of bounds or over a location where an ant already exists.	occur, board just looks the same for the given number of steps.	possible. Instead, ran 10x10 grid with 1 ant and 1 doodlebug for 100 steps (with valgrind).	exit all heap blocks freed, no issues.
Run iterations with one ant for ~15 steps, fair sized board.	Ensure that ant breeding works properly (breeding every 3 steps) and that ants keep multiplying.	Ants should breed every three steps. On a 15-step run there should be 32 ants (2^5) with no predators on the board. Large board for test conditions will hopefully prevent condition where ants fail to breed.	10x10 grid, 1 Doodlebug and 1 Ant initialized, 15 steps. Re-tried a couple of times until Ant and Doodlebug spawned sufficiently far apart.	Doodlebug died on third step, and Ant multiplied on third step.
Run iteration with only doodlebugs for 3 steps	Ensure that doodlebug starve mechanism works (death after 3 steps with no ants eaten)	All doodlebugs should die.	Due to design this exact scenario was not possible, instead ran 10x10 board with 99 Doodlebugs and 1 Ant. Expected all but 1 Doodlebug to die on step 3, then last Doodlebug to die on step 4	Ran as expected.
Run iteration completely full of doodlebugs for 10 steps.	Ensure that doodlebug starve mechanism works, that doodlebugs do not move outside of bounds or over other doodlebugs.	Board will be full of doodlebugs for 3 turns and then they will all die.	Ran with valgrind in a similar manner to last test, but with 20x20 grid, 1 ant, and 399 Doodlebugs for 10 steps.	All but 1 Doodlebug died on step 3, last one died on step 4, blank board for the rest of the game. Prior to Doodlebug exodus the blank space did move, indicating Doodlebug movement where possible. No valgrind errors.

Run iteration nearly full of ants and one doodlebug for 25 steps	Ensure that doodlebug breeding works properly (breeding every 8 steps) and that doodlebugs keep multiplying.	Doodlebugs should breed every 8 steps. With sufficient ants to avoid doodlebug starvation at the start, running 25 times should result in there being 8 doodlebugs at the end.	30x30 board, 1 Doodlebug, 899 Ants, increments of 8 steps.	Second Doodlebug spawned at step 8, third and fourth spawned at 16, 5-8 at step 24, 8-16 at step 32. Doodlebug spawning looks to be working to spec.
Run iteration with full board, half ants and half doodlebugs	Ensure program places all critters so board is full - every space must be filled with no attempted overwrites or out-of-bounds issues.	All spaces are filled at beginning, program runs as specified.	40x40, 800 Doodlebugs and 800 Ants, 3 steps	It was a massacre. All Ants consumed by step 7 with Doodlebug extinction soon to follow.
Run with valgrind.	Check for memory leaks.	No memory leaks.	Ran several of the above tests utilizing valgrind	No leaks or errors detected.

Reflection:

Working in a group, especially exclusively online, presented challenges and issues beyond the normal scope of a coding project. Since the segments of the project were so divided, it made testing code and ensuring it works a lot more difficult than if you are the sole author of an entire project. Our group worked well regarding communication and expectations of roles, which benefitted us. The project would have been significantly harder if the communication between team members was lacking.

The strategy of having a group voice chat and dividing things out with hard deadlines for early draft versions proved to be successful. Manda and Phillip were able to communicate with Derth regarding the interface of the Grid and Doodlebug/Ant classes. Yuxing provided a handy input validation function that everyone utilized, and Kayla was able to test and piece everything together. We primarily used Slack to communicate after our first voice chat, and that worked well. We ran into no substantial issues during the course of the project.

CS 162 Group Project Initial Design
Derth Adams
April 26, 2019

I've gone through the assignment requirements and come up with an initial design. The process was a lot harder than I thought and my solution is less elegant than I hoped, because of one central problem:

We're required to use a dynamic array of pointers to Critter to represent the grid (which should probably go inside a Grid class which also contains the logic for triggering updates on each Ant/Doodlebug.).

We're also required to have all of the logic for the movements of individual Ants/Doodlebugs inside the virtual `move()` function in the Critter class, which is overridden in Ant and Doodlebug.

In order for an Ant/Doodlebug to determine where it's supposed to move, it needs to have some information about its immediate surroundings. It needs to know if each pointer in the grid immediately above, below and to each side of it are null, point to an ant, or point to a doodlebug (or if it's on the edge/corner of the grid and can't move in certain directions).

Also, in order for the Ant/Doodlebug to move, its address needs to be copied to a different pointer in the 2D array.

I think the big design question is, how does the Ant/Doodlebug get access to the 2D pointer array to make decisions about what happens next, and how does it move itself, delete other objects, or delete itself? Since an Ant or Doodlebug object exists inside the Grid object, it can't naturally call that object's member functions, since it has no natural access to the object instance.

One of the solutions I tried was for the Grid object, when it was updating an Ant/Doodlebug, to code the states of the four squares surrounding it into an array of four enums and pass it to the Ant/Doodlebug in the `move()` call. The Ant/Doodlebug would look at the states of the four squares, figure out what it needed to do, then set some internal flags (like "moved", "ate", "starved"). After `move()` returned, Grid would check all those flags and implement all the changes (like moving/deleting objects). This felt sort of clunky, though.

Then I thought that it might be simpler to have Grid pass a pointer to itself to the Ant/Doodlebug (by passing "this" in the `move()` function). This would allow the Ant/Doodlebug to call Grid methods, and if Ant and Doodlebug were declared friend classes of Grid, they could also get direct access to the 2D array, and could move themselves from pointer to pointer and delete other objects as necessary. I didn't think an object should delete itself, so instead of having a Doodlebug delete itself when it starves, I had it set a starve flag and Grid does the deletion.

That's the solution I built out in pseudocode - I keep thinking there must be a better way to do it, so if someone else finds a better solution that's great!

The following section is higher-level pseudocode to show how this solution could be implemented. I have a more detailed version, but it was way too long so I turned most of the function descriptions into higher-level summaries, then went into more detail in `Ant::move()` and `Doodlebug::move()`, where the bulk of the important logic takes place.

enum State {empty = 0, ant = 1, doodlebug = 2, offGrid = 3}

Used for two purposes:

To pass information about an Ant/Doodlebug's immediate surroundings from the getNeighbor() member function of the Critter class.

As a member variable of the Critter class to represent the subclass that the Ant/Doodlebug belongs to.

enum Compass {North = 0, East = 1, South = 2, West = 3}

Represents the four cardinal directions - used in several different member functions of Critter.

GAME class:

This class contains all of the game logic and all of the other objects in the game, like the Grid and the Ants/Doodlebugs.

Member variables:

int Steps = number of steps the simulation will run

Grid* grid = a pointer to the Grid object containing all the Critters

Member functions:

void playGame():

This function gets all of the initial game parameters from the user, dynamically instantiates a Grid object with the desired parameters, runs the simulation for the desired number of steps, then asks the user if they want to continue or quit.

If they want to continue, gets the additional number of steps and continues running the simulation. If they want to quit, the function returns.

void playStep():

This function is responsible for running a single step in the simulation. Updates all the Ants/Doodlebugs on the grid by calling grid.updateCritters(), prints the current grid state by calling grid.print(), and increments Step by 1.

~Game()

Deallocates the Grid object

GRID class:

Declare Ant and Doodlebug as friend classes (so they can directly access Critter*** array)

Member variables:

int sizeM = vertical size of the grid

int sizeN = horizontal size of the grid

(I'm using matrix-style indexing, so the M and N indices start at 0 in the upper-left corner of the 2D array. M increases as it goes down, N increases as it goes right.)

Critter* array** = Points to a dynamically allocated 2D array of pointers to Critter which represents the grid in which the Ants/Doodlebugs live

Member functions:

Grid (int sizeM, int sizeN, int numAnts, int numDoodle)

*Sets the grid's dimension variables sizeM and sizeN to the input parameter values
Dynamically allocates a 2D array of pointers to Critter of the desired size*

*Populates the grid with the desired number of randomly placed ants
and doodlebugs*

~Grid()

Deallocates the memory used for the 2D array

bool addAnt (int coordM, int coordN)

*Tries to add an ant to the array at the specified coordinates. Returns true if successful,
false if not.*

bool addDoodlebug (int coordM, int coordN)

*Tries to add a doodlebug to the array at the specified coordinates. Returns true if
successful, false if not.*

void resetCritters()

Sets every Ant/Doodlebug's status flags to false by calling reset() on each object.

void updateCritters()

*Reset all ant/doodlebug flags using resetCritters()
Update the doodlebugs by calling move(this) on each doodlebug object in the array.
If a doodlebug starved during the update, remove it from the array.
Update the ants by calling move(this) on each ant object in the array*

void print()

*Prints out the Grid using "X" for doodlebugs, "O" for ants, and " " for unoccupied
Squares*

CRITTER class:

This is the superclass of Ant and Doodlebug.

Member variables:

State critterType = the subclass of Critter the object belongs to (ant or
doodlebug)

int coordM = vertical coordinate of critter's current position

int coordN = horizontal coordinate of critter's current position

int breedCounter = integer representing the number of steps since the critter
last bred

bool updated = set to true if the critter was updated during the current step

Member functions:

Critter (int coordM, int coordN)

Sets the critter's current position to coordM and coordN

State getType()

Returns the critter's critterType (ant or doodlebug)

virtual void move(Grid* grid)

*Contains the logic for an Ant/Doodlebug's activities during one step of the simulation.
Overridden in the Ant and Doodlebug classes.*

virtual void breed(Grid* grid, Compass direction)

Overridden in Ant and Doodlebug

Compass getRandomDirection()

Returns a randomly selected Compass enum (North, East, South, or West)

void moveTo (Grid* grid, Compass direction)

Moves the Ant/Doodlebug one square over in the grid in the specified direction, and updates its coordinates to those of the new square.

void shiftCoords(int& destM, int& destN, Compass direction)

Takes two ints by reference representing M and N coordinates of a square, and changes them so that they point to the next square over in the specified direction.

State getNeighbor(Grid* grid, Compass direction)

Returns the state of the square adjacent to the ant/doodlebug in the specified direction.

virtual void reset()

Sets the "updated" flag to false (overridden in Doodlebug)

ANT class (inherits from Critter)

Member variables:

Member functions:

Ant (int coordM, int coordN)

*Calls the Critter constructor with the input parameters.
Sets critterType to ant.*

void move(Grid* grid) override

Choose a random direction using getRandomDirection()

Get the state of the neighboring square in that direction using
getNeighbor(grid, direction)

If the square is empty:

Move to that square using moveTo(grid, direction)

Increment breedCounter by 1

Else if the square is not empty:

Increment breedCounter by 1

If breedCounter == 3:

Set a counter to 4

Get a random direction using getRandomDirection()

While the square in the random direction is not empty and counter > 0:

Move on to the next direction clockwise

Decrement the counter by 1

If the chosen square is empty:

Give birth to a new ant in the square using breed(grid, direction)

Reset breedCounter to 0

void breed(grid, direction) override

Add a new Ant object in the square adjacent to the current ant in the specified direction.

DOODLEBUG class (inherits from Critter)

Member variables:

bool starved = set to true if Doodlebug starved during the current update (allows the Grid object to delete the Doodlebug object at the end of the update)

bool ate = set to true if Doodlebug ate during current update

int eatCounter = integer representing the number of steps since the Doodlebug last ate

Member functions:

Doodlebug (int coordM, int coordN)

Calls the Critter constructor with the input parameters.

Sets critterType to doodlebug.

void move(Grid* grid) override

For each Compass direction, moving clockwise from North to West:

Get the state of the neighboring square in that direction using
getNeighbor(grid, direction)

If the square contains an ant && ate == false:

Eat the ant using eatAnt(grid, direction)

Set ate = true

Reset eatCounter to 0

If ate == false:

Get a random direction using getRandomDirection()

If the neighboring square in that direction is empty:

Move doodlebug to that square using moveTo(grid, direction)

Increment eatCounter by 1

If the neighboring square in that direction is not empty:

Increment eatCounter by 1

If eatCounter == 3:

Set starved = true

If starved == false:

Increment breedCounter by 1

If breedCounter = 8:

Set a counter to 4

Get a random direction using getRandomDirection()

While the square in the random direction is not empty &
the counter > 0:

Move on to the next direction clockwise

Decrement the counter by 1

If the randomly chosen square is empty:

Give birth to a new doodlebug in the chosen
square using breed(grid, direction)

Reset breedCounter to 0

void eatAnt(Grid* grid, Compass direction)

Delete the ant in the square adjacent to the doodlebug in the specified direction, move the current doodlebug to that square, and update the doodlebug's coordinates to those of the new square.

void breed(Grid* grid, Compass direction) override

Add a new Doodlebug object in the square next to the current doodlebug in the specified direction.

bool hasStarved()

return starved

void reset() override

starved = false

ate = false

updated = false