Group Project Design + Reflection

Program Design

Problem

In the Predator-Prey game, we will use cellular automata to create a 2D predator–prey simulation in your program. The preys are ants and the predators are doodlebugs.

Predator-Prey Game Requirements

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.

Input Validation:

We will recycle our InputValidation class used in Project 1. It has an overloaded function called getInt(). We chose to make InputValidation its own class so that it could be used in future programs. getInt() receives input from the user, converts it into a stringstream, which then checks to see if there is anything beyond an integer that is left over. If there is, it makes the user try to input an integer again.
getInt(int min, int max) serves the same function as getInt() but also runs another validation checking input to see if it lies between min and max. getInt(min, max) will be useful for specifying minimum and maximum values for the following inputs:

- Initial settings, such as: number of steps, rows, columns, ants, and doodlebugs
- Menu options for continuing the simulation or exiting the program

Menu:

We will recycle our Menu class used in Project 1 to display menu options, get input from the user, and store that input as a user choice variable. We chose to make Menu its own class so that it could be used in future programs. It references our Input Validation class to ensure the user is entering a valid input from the menu. There are functions to add an option to the menu, print the menu options, ask the user for his/her choice, and return the user choice.

The only menu is a continue menu prompting the user to continue the simulation or exit the program.

Critter Class:

The Critter class contains data and functions common to ants and doodlebugs.

The Critter class will contain the following variables: row position, column position, and an enum variable representing the critter's orientation (UP, RIGHT, DOWN, LEFT). The enum orientation variable will be useful when picking a random direction for each critter to move.

This class will have a virtual function named move that is defined in the derived classes of Ant and Doodlebug. Thus, the Critter class is an abstract base class. It will also have a virtual function named breed defined in Ant and Doodlebug.

Each Critter's position will be set using a constructor with 2 int parameters representing the starting row and column. There will be a getBugType function that returns a char representing an ant or doodlebug as 'O' and 'X' respectively. Lastly, there will be a bool variable hasMoved that represents whether a Critter has moved in a step in order to avoid a double movement while cycling through our dynamic array pointing to Critter objects. For example, a critter could move to a row or column greater than the current, then be told to move again in a future iteration of the move step loop resulting in 2 movements when there should only be one.

Ant Class:

Requirements:
The ants behave according to the following model:

- Move: For every time step, the ants randomly move up, down, left, or right. If the neighboring cell in the selected direction is occupied or would move the ant off the grid, then the ant stays in the current cell.
- Breed: If an ant survives for three time steps (not been eaten by doodlebugs), at the end of the time step (i.e., after moving) the ant will breed. This is simulated by creating a new ant in an adjacent (up, down, left, or right) cell that is empty randomly. If the cell that is picked is not empty, randomly attempt to pick another one. If there is no empty cell available, no breeding occurs. Once an offspring is produced, an ant cannot produce an offspring again until it has survived three more time steps.

Design:
The ant class was designed with the above criteria in mind. It only contains two main functions: move and breed.
Move first increases the breedCounter variable representing how many days it has been since it breeded. A random direction is chosen by selecting a random number between 0 and 3 which correspond to our Orientation enum variable values of {UP, RIGHT, DOWN, LEFT} from the Critter class. For each of the directions, Ant will check to see if there is room to move in that direction (not at the edge) and if the space is currently occupied by another critter. If there is room to move in the direction and there is no critter, it will then move to that position. Our hasMoved variable will then switch to true.
Breed checks the breedCounter variable to see if 3 steps have passed. If so, each direction is tested to see if

there is an empty space to breed. If there's a valid space to breed, a random direction is selected from available options, and a new ant is placed in that direction.

Doodlebug Class:

Requirements:
The doodlebugs behave according to the following model:

- Move: For every time step, the doodlebug will firstly try to move to an adjacent cell containing an ant and eat the ant (you can decide if there are several ants in the adjacent cells, how the doodlebug will choose to move). If there are no ants in adjacent cells, the doodlebug moves according to the same rules as the ant. Note that a doodlebug cannot eat other doodlebugs.
- Breed: If a doodlebug survives for eight time steps, at the end of the time step, it will spawn off a new doodlebug in the same manner as the ant (the Doodlebug will only breed into an empty cell).
- Starve: If a doodlebug has not eaten an ant within three time steps, at the end of the third time step it will starve and die. The doodlebug should then be removed from the grid of cells.

Design:

Doodlebug is quite similar to Ant in terms of functions although the implementations of the functions are different.
Move checks surrounding spaces for ant targets and adds them to a vector of possible moves. If there is a valid ant target to eat, doodlebug:

1. selects one at random
2. deletes the existing ant object
3. moves the doodlebug to the new position
4. clears the old position
5. updates coordinates
6. sets hasMoved variable to true
7. updates starveCounter.

Otherwise, there are no ant to eat, it moves normally exactly like an Ant moves as described above.

Breed checks to see that the Doodlebug has been alive for 8 steps, and if so functions identically to how an Ant breeds as described above.

Game Class:

Requirements:
For each time step, do the following in your program: after moves, when breeding, eating, and starving are resolved, display the resulting grid. 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). The doodlebugs will move before the ants in each time step. When you reach the time steps entered by the user, ask them to enter another number and start to run the simulation again or to exit the program. You must maintain the state of the current grid while creating the next display.

You should use a dynamic array to represent the grid. Each array element will be a pointer to a Critter. Get your program running and tested. You should see a cyclical pattern between the population of predators and prey, although random perturbations may lead to the elimination of one or both species.

We also chose to pursue extra credit which requires:
In addition to prompting the user for the number of time steps, ask them to enter the **size of the grid rows and columns, the number of ants, and the number of doodlebugs**. If you did the extra credit part, **you must print a line on the screen at the beginning of your program to inform the grader that you did it**.

Design:
Game will create a dynamic array of Critter pointers using a triple pointer variable 'board'. There is a function createBoard that will allocate the board based on rows and cols member variables. The deallocation will occur via the ~Game destructor. There will also be a function called printBoard in order to print the board array of Critter pointers, representing ants as 'O', doodlebugs as 'X', and blank spaces as a space.

There are some functions for getting initial game settings, such as setTimeStep, setRows, setCols, setAntCount, and setDoodleCount. Each of these functions utilizes the Menu/InputValidation classes in order to receive valid data for the corresponding variables. We must also address the fact that a small board cannot hold more critters than the board could hold.

Once all of these settings are received, critters will be placed via the placeCritters function which randomly places them on the board based on ant and doodlebug numbers received by the user.

Once the board is set, we will have one driver function called makeMoves that calls on several other functions in order to satisfy the game flow specified in the specifications. The game flow, as stated above, should follow the following sequence of events(which will be our function names): moveAllDoodlebugs, starveAllDoodlebugs, breedAllDoodlebugs, moveAllAnts, breedAllAnts, and resetHasMoveFlags. Each one of these functions is aptly named to describe what they do.

Main:

main creates the board after getting desired settings from the user. After retrieving the settings for the simulation, main runs a do-while loop running the simulation, setting the next number of steps for a continued simulation, and prompting the user whether to run the simulation or quit the program.

Reflection:

We approached this program methodically starting with importing work from previous projects: our already written Menu and InputValidation classes. We also took what relevant code we could from other projects such as Langton's Ant and our OSU University system involving virtual functions. We found that starting out with these small victories gave us confidence when the project looked so daunting at first.

We chose a "divide and conquer" approach to this project based on everyone's perceived strengths and weaknesses. Our InputValidation/Menu classes were compared to see which might be most adaptable for this project. Kevin's InputValidation/Menu classes were used. Kevin started working on a shell of the Critter class just to help get the project started. Ann took on the Doodlebug class, Kyle took the Ant class, and Eric took the Game class. John was in charge of testing and helping the others as needed. Kevin volunteered to do the documentation since the Critter class was not too involved.

In theory, the divide and conquer approach seems to be most efficient, but in practice, our group felt that it might have led to a collection of disjointed files that did not cooperate well with each other. In fact, when we finally got to a point where we each felt like our code was usable, we compiled it together only to find numerous errors as a result. The errors were so thick that Kevin felt like we could not get through them in a reasonable amount of time so he decided to start over piecing everyone's code together, being sure to communicate with the team along the way. He took everyone's code piece by piece and unit tested as he went. Does the board print? Yes, move on. Do ants appear on the board? Yes, move on. Do the ants move correctly? Etc.

With this approach we got back to a point where we could again make progress to finish the program out. The others in the team stepped up in a large way as the deadline approached and we began to share the program more instead of working in relative isolation. We believe that this team effort and collaboration led us to our successful outcome.

In the future, we believe constant communication to be key to our success. We also consider using a more streamlined approach to avoid the disjointedness, such as using git. We did not use git for this project because not all members of the group were familiar or comfortable using it. We feel that this would be a worthwhile language to learn in order to minimize inefficiencies in future projects.

Test Tables

For InputValidation Class – This class was already tested in Project 1

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
input too low or high	input < 2 input > 200	getInt(2, 200)	You must select an integer equal to or between 2 and 200.	You must select an integer equal to or between 2 and 200.
input in correct range	2 < input < 200	getInt(2, 200)	Returns input	Returns input
input extreme low	input = 2	getInt(2, 200)	Returns 2	Returns 2
input extreme high	input = 200	getInt(2, 200)	Returns 200	Returns 200

input not an integer	input = a	getInt(2, 200)	You must enter an integer value.	You must enter an integer value.
input not an integer	input = 123abc	getInt(2, 200)	You must enter an integer value.	You must enter an integer value.
input not an integer	input = enter key	getInt(2, 200)	You must enter an integer value.	You must enter an integer value.
input not an integer	input = asdf 123	getInt(2, 200)	You must enter an integer value.	You must enter an integer value.

For Menu Class - This class was already tested in Project 1

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes

input option string	"Start Langton\'s Ant simulation"	addOption(string)
		printOptions()
		main()
			1. Start Langton's Ant simulation	1. Start Langton's Ant simulation

input option string	"Quit"	addOption(string)
		printOptions()
		main()
			1. Start Langton's Ant simulation
			2. Quit
				1. Start Langton's Ant simulation
				2. Quit

input too low or high	input < 1
	input > 2
		getInt(1, 2)
		promptUserChoice()
		main()
			You must select an integer equal to or between 1 and 2.	You must select an integer equal to or between 1 and 2.

input not an integer	input = a	getInt(1, 2)
		promptUserChoice()
		main()
			You must enter an integer value.	You must enter an integer value.

input valid	input = 1
	input = 2
		getInt(1, 2)
		promptUserChoice()
		getUserChoice();
			returns 1
			returns 2
				returns 1
				returns 2

For Game Class

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes

board is created and printed properly at various square sizes	rows = 2
	cols = 2
	rows = 20
	cols = 20
	rows = 200
	cols = 200
		createBoard
		setRows
		setCols
		printBoard
		main
			2x2 empty array is created and printed to console
			20x20 empty array is created and printed to console
			200x200 empty array is created and printed to console
				2x2 empty array is created and printed to console
				20x20 empty array is created and printed to console
				200x200 empty array is created and printed to console

board is created and deallocated properly at various square sizes	rows = 2
	cols = 2
	rows = 20
	cols = 20
	rows = 200
	cols = 200
		createBoard
		setRows
		setCols
		~Game
		main
			2x2 empty array is created and deallocated
			20x20 empty array is created and deallocated
			200x200 empty array is created and deallocated
			NO SEG FAULTS!
				2x2 empty array is created and deallocated
				20x20 empty array is created and deallocated
				200x200 empty array is created and deallocated
				NO SEG FAULTS!

board is created and printed properly at various rectangular sizes	rows = 2
	cols = 20
	rows = 20
	cols = 2
	rows = 100
	cols = 200
	rows = 200
	cols = 100
		createBoard
		setRows
		setCols
		printBoard
		main
			2x20 empty array is created and printed to console
			20x2 empty array is created and printed to console
			100x200 empty array is created and printed to console
			200x100 empty array is created and printed to console
				2x20 empty array is created and printed to console
				20x2 empty array is created and printed to console
				100x200 empty array is created and printed to console
				200x100 empty array is created and printed to console

board is created and deallocated properly at various rectangular sizes	rows = 2
	cols = 20
	rows = 20
	cols = 2
	rows = 100
	cols = 200
		createBoard
		setRows
		setCols
		~Game
		main
			2x20 empty array is created and deallocated
			20x2 empty array is created and deallocated
			100x200 empty array
				2x20 empty array is created and deallocated
				20x2 empty array is created and deallocated
				100x200 empty array is created and deallocated

	rows = 200
	cols = 100
			is created and deallocated
			200x100 empty array is created and deallocated
				200x100 empty array is created and deallocated

ants and doodlebugs placed randomly in the correct amounts on the board	rows = 50
	cols = 50
	ants = 5
	doodlebugs = 1
	ants = 20
	doodlebugs = 5
	ants = 5
	doodlebugs = 100
		setAntCount
		setDoodleCount
		placeCritters
		createAnt
		createDoodlebug
			5 ants and 1 doodlebug placed randomly on game board
			20 ants and 5 doodlebug placed randomly on game board
			5 ants and 100 doodlebug placed randomly on game board
				5 ants and 1 doodlebug placed randomly on game board
				20 ants and 5 doodlebug placed randomly on game board
				5 ants and 100 doodlebug placed randomly on game board

Number of ants and doodlebugs cannot exceed available spaces on the board	rows = 20
	cols = 20
	ants = 400
	doodlebugs = 200
	ants = 200
	doodlebugs = 400
		setAntCount
		setDoodleCount
		placeCritters
		createAnt
		createDoodlebug
			setAntCount allows 400 ants but setDoodleCount will not allow 200 doodlebugs to be created as there are only 400 spaces total
			setAntCount allows 200 ants but setDoodleCount will not allow 400 doodlebugs to be created as there are only 400 spaces total
				setAntCount allows 400 ants but setDoodleCount will not allow 200 doodlebugs to be created as there are only 400 spaces total
				setAntCount allows 200 ants but setDoodleCount will not allow 400 doodlebugs to be created as there are only 400 spaces total

makeMoves controls gameplay correctly	varies	makeMoves moveAllDoodlebugs starveAllDoodlebugs breedAllDoodlebugs moveAllAnts breedAllAnts resetHasMovedFlags	Doodlebugs eat surrounding ants first, move to empty spaces if there are no ants to eat, starve if they haven’t eaten in 3 steps, and then breed if they have been alive for 8 days. Ants move to an empty	Doodlebugs eat surrounding ants first, move to empty spaces if there are no ants to eat, starve if they haven’t eaten in 3 steps, and then breed if they have been alive for 8 days. Ants move to an empty adjacent space, and breed

			adjacent space, and breed if they have been alive for 3 steps	if they have been alive for 3 steps

For Critter/Ant Classes

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Ant moves in a random direction	random direction value using enum Orientation	move	Ant moves in a random direction on the game board	Ant moves in a random direction on the game board
Ant does not move to another space that is occupied	Large number of ants on a board	move	Ant moves to another space that is not occupied only	Ant moves to another space that is not occupied only
Ant cannot move off edge of the board	Ants placed at edge of board Direction set to move towards edge	move	Ants when directed to move off the board, do not do so and stay at the same space.	Ants when directed to move off the board, do not do so and stay at the same space.
Ants breed after 3 days	Ants placed on board randomly	breed	Ants create a new ant on a random adjacent space after 3 steps	Ants create a new ant on a random adjacent space after 3 steps

For Critter/Doodlebug Classes

Test Case	Input Values	Driver Functions	Expected Outcomes	Observed Outcomes
Doodlebug moves in a random direction	random direction value using enum Orientation	move	Doodlebug moves in a random direction on the game board	Doodlebug moves in a random direction on the game board

Doodlebug eats a surrounding ant	Ants and doodlebugs put randomly onto game board	move	Doodlebug eats a surrounding ant by moving to the adjacent ant space and deleting the ant object	Doodlebug eats a surrounding ant by moving to the adjacent ant space and deleting the ant object
Doodlebug eats a surrounding ant before moving to an empty space	Ants and doodlebugs put randomly onto game board	move	Given the choice of eating an adjacent ant or moving to an empty space, doodlebug moves to eat the ant in all cases	Given the choice of eating an adjacent ant or moving to an empty space, doodlebug moves to eat the ant in all cases
Doodlebug does not move to another space that is occupied	Large number of doodlebugs on a board	move	Doodlebug moves to another space that is not occupied only	Doodlebug moves to another space that is not occupied only
Doodlebug cannot move off edge of the board	Doodlebugs placed at edge of board Direction set to move towards edge	move	Doodlebugs when directed to move off the board, do not do so and stay at the same space.	Doodlebugs when directed to move off the board, do not do so and stay at the same space.