Predator-Prey Game Design Document

Author: Solomon Deutsch
Eric Morgan
Steve Perreault
Bret Swalberg
Chen Zou

PROJECT DESIGN

Program Flow

1. Prompt for user input on the following:
   - The number of row and columns to construct the board.
   - How many turns the game should last for.
   - Number of ants and doodlebugs to start.
2. Prompt user to choose the run mode:
   - Choice 1 - Run continually
   - Choice 2 - run one step at a time
3. If choice 1 in step 2 was chosen, prompt the user for how many milliseconds the program should pause after the program displays each board.
4. Place each ant and doodlebug at a randomly generated location. Display the initial placement of all critters to the user.
5. Start running the game, repeat the following steps for each turn until the user specified number of turns has been reached.
   - Add 1 to the age of each critter on the board;
   - Choose the doodlebug that is the first in the sequence (starting from top left and ending at the bottom right). Identify any north, south, west, or east squares that have an ant on them. If there are any such squares, choose one of the squares randomly. Move to it and remove the ant. If there are no such squares, choose north, south, east or west randomly. If the square in the chosen direction is unoccupied, move to it. Otherwise, stay put. Continue this process for all of the doodlebugs.
   - Choose the ant that is the first in the sequence (starting from top left and ending at the bottom right). Move that ant to the north, south, west, or east square, chosen randomly. If the randomly chosen square is occupied, stay put. Continue this process for all of the ants.
   - Doodlebugs breed- check each Doodlebug starting at the top left. If it bred 8 days ago or if this is its 8th turn, check the 4 adjacent squares and place a new doodlebug on one of those squares, if vacant, at random. Or, if only one square if vacant, then on the one square. Continue through the whole board, repeating this process for each doodlebug.
   - Go through the same breeding process for ants, except base the breeding ability on whether the ant bred 3 days ago or started its first turn 3 days ago.
   - Go through the board, and remove any doodlebug that has not eaten an ant in this turn and has not eaten an ant in the last 2 turns.

- Display the board with all of the critters on it. The border of the board is represented by asterisks, the ants are represented by O, the doodlebugs are represented by X, available spaces are represented by whitespace.
- Prompt the user to press enter to simulate the next turn.

6. When all turns have been simulated, ask the user if they would like to continue the game by indicating the number of extra terms they would like to simulate. Also notify the user that they can choose 0 turns to quit.
7. After the user has chosen 0 additional turns, prompt them to press enter to exit the program.

Classes Required

Game Class:

private:

```
Board* board;                   // pointer to the game board
Critter** doodlebugs;           // array of pointers to all ants
Critter** ants;                 // array to pointers to all doodlebugs
short int numberSteps;          // number of steps game should run for
short int currentStep = 1;      // the current step
short int numAntsStart;         // the starting number of ants
short int numDoodlebugsStart;   // the starting number of doodlegus
short int runMode;              // 1=Run continuously, 2=Run step-by-step
short int continualModeSleep;   // Time to sleep during each continuous iteration
```

public:

```
Game(short int nSteps, short int nCols, short int nRows, short int numAntsStart,
        short int numDoodlebugsStart, short int nRunMode, short int nContinualModeSleep);
~Game();
void init();
void run();
void runTurn();
void moveCritters();
void breedCritters();
void ageCritters();
void handleStarvation();
void placeCritter(CritterType type, short int numberToPlace);
static int calculateMovement(Board* board, int currentSpace, int moveDirection);
static std::string getDirectionString(int direction);
```

Board Class:

private:

```
const char ANT_SPACE = 'O';    // characters that represent the ant occupied spaces
const char DOODLEBUG_SPACE = 'X';   // doodlebug occupants of board spaces
const char EMPTY_SPACE = ' '; // empty spaces
const char BORDER = '*';          // border does not occupy any space on the board, instead it is
                                     drawn around the board by the displayBoard() method
```

```
short int rows;                 // size of the board in rows
short int columns;              // size of the board in arrays
int adjacentSpaces[4];          // array which holds the content of the adjacent
                                // spaces relative to a given location
                                // 0 = north, 1 = south, 2 = east, 3 = west
Critter** boardSpaces;          // pointer to an array of pointers to critter objects

public:
Board(short int nRows, short int nColumns);
~Board();
short int getRows() { return rows; };
short int getColumns() { return columns; };   // methods needed for working with the game board
short int getBoardPosition(short int row, short int column) const;
bool placeCritter(Critter* critter, short int position);
void moveCritter(short int fromSpace, short int toSpace);
void removeCritter(short int position);
Critter* getBoardCritter(short int position) const;
short int getRemainingSpaceCount() const;
short int* getRemainingSpaces(short int & totalSpaces) const;
short int* getCritterSpaces(CritterType, short int &) const;
int* getAdjacentSpaces(int position);
void displayBoard() const;
```

Critter Class and derived Classes (Doodlebug and Ant):

The Character Class is an abstract class with 5 derived classes, the hierarchy diagram is illustrated in the Figure 1 below.

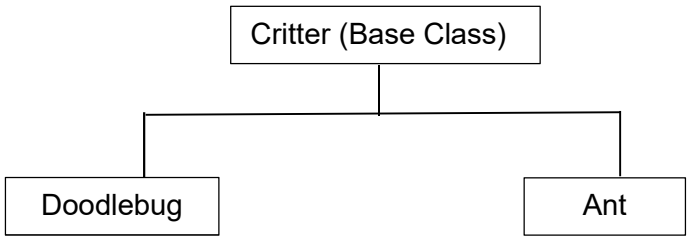

Figure 1 – Class Inheritance Hierarchy

The members of the class are listed in the table below:

Class	**Critter**	**Doodlebug**	**Ant**

Potected for base
 	
Private for derived
	const CritterType critterType; // holds the name of critter type (Ant or Doodlebug)
	short int age = 1; // all critters start with age of 1 step
		int starvingDay; // the continuously starving day of doodlebug	

	short int breedingInterval; // how often does critter reproduce		

Public	Critter(CritterType critterT) : critterType(critterT) { };
	virtual ~Critter();
	
	// getter and setter methods
	CritterType getCritterType() { return critterType; };
	short int getAge() { return age; };
	void setAge (short int age) { this->age = age; };
	short int getBreedingInterval() { return breedingInterval; }
	virtual int move(int pos, int* adjacentSpaces) = 0;
	int breed(int currentSpace, int* adjacentSpaces);
	void incrementAge();
	std::string getCritterName();
		Doodlebug();
		~Doodlebug();
		int move(int currentSpace, int* adjacentSpaces);
		void setStarvingDay(int);
		int getStarvingDay();
			Ant();
			~Ant();
			int move(int currentSpace, int* adjacentSpaces);

Key Algorithms

Place Critters:

1. Get the remaining empty spaces.
2. Limit the maximum number to place to the total available spaces or given number passed as argument, whichever is less.
3. Place the defined number of critters to the board. For each critter, repeat the following:
   - Get pointer to array of empty spaces, as well as the total spaces available to place critters
   - Get random number representing random position at which to place critter
   - Create a new critter based on the critter type passed as argument.
   - Mark the placed status.
   - Delete the pointer to the empty spaces.

Move Critters:

1. Get pointer to array containing locations of all doodlebugs.
2. Loop through total spaces containing doodlebug, do the following for each space:
   - Get adjacent space to move to - Identify any north, south, west, or east squares that have an ant on them. If there are any such squares, choose one of the squares randomly. Move to it and remove the ant. If there are no such squares, choose north, south, east or west randomly. If the square in the chosen direction is unoccupied, move to it.
   - Get the critter on the space and get the move direction
   - Calculate the new space based on the move direction
   - Move critter from the current space to the new space

- Loop to the next position

3. Reset the position index to zero and repeat the above steps for all of the ants, for ant movement, the destination is an adjacent location randomly picked from north, south, west, or east. If the randomly chosen square is occupied, ant will stay.
4. Delete the pointers for antSpaces and doodlebugSpaces to free dynamically allocated memory and reset the pointers.

Note: when a critter is trying to move, it it's against the edge of any part of the board, then the board walls are considered unavailable spaces for the critter and the critter will not move.

Breed Critters:

1. Get pointer to array containing locations of all doodlebugs
2. Loop through total spaces containing doodlebug, do the following for each space:
   - Get adjacent space to breed to - choose north, south, east or west randomly;
   - Check if breeding interval has been met. If it bred 8 days ago or if this is its 8th turn, check the adjacent squares;
   - If no empty adjacent cells, no breeding. Else keep checkinging random adjacent cells until an empty one if found;
   - Create a new doodlebug and place to the empty adjacent cell found in the above step;
   - Loop to the next position.
3. Delete the array of doodlebug spaces to free memory and reset pointers to null;
4. Reset the position index to zero and repeat the same breeding process for ants, except base the breeding ability on whether the ant bred 3 days ago or started its first turn 3 days ago.
5. Delete the array to antSpaces to free dynamically allocated memory and reset the pointers.

Note: If the ant or the doodlebug were unable to breed, the critter still has to wait another 3 or 8 turns before it can try to breed again.

Handle Starvation:

1. Get pointer to array containing location of all doodlebugs
2. Iterate over the locations in the array, for each doodlebug:
   - Create a temporary pointer to doodlebug on the current space.
   - Check if the doodlebug has been starving for 3 days, remove the doodlebug if so
   - Delete the temporary pointer to the doodlebug
   - Continue to the next position

TESTING PLAN

The program has been thoroughly tested and any problems and imperfections revealed were fixed. The testing cases are summarized in the table overleaf.
The Note: we tested with more input values than indicated in this table. This table is simply designed to capture the essence of each test.

Test Case	Input Values	Built in functions assisting in test	Expected Outcomes	Actual Outcomes
No memory leaks	Ran several of the below tests with valgrind		All allocated memory is freed, no errors.	Performed as expected.
Board size is handled correctly	21 columns, 19 rows	n/a	Borders of board should have 23 asterisks at the top and bottom and 21 asterisks at the right and left.	Performed as expected.
Critter placement	13 x 18 board, 200 ants, 12 doodlebugs	getCritterCount in board class	212 critters on initial board	Fixed now
Input validation for number of rows, columns, ants, doodlebugs, moves	Enter 15a, @dfe, twenty. 4 columns, 5 rows, 21 ants.	n/a	Reprompt user for appropriate info. Prompt user for input within range.	Performed as expected. Fixed now
Correct number of ants and doodlebugs on the board after the move.	22 columns, 17 rows, 100 ants, 3 doodlebugs, run 2 turns.	db (standalone) function	Number of critters should be 103 - number of ants that were eaten on turn 1 and 2.	Fixed now
Breeding occurs at appropriate times.	Run simulation for 24 turns	Db (standalone) function	If critters of a type are alive, there should be attempted breeding for ants on turn numbers that are a multiple of three. Likewise, there should be attempted breeding for Doodlebugs on turn numbers that are a	Fixed now.

			multiple of eight.	
Population grows in a mathematically possible/ reasonable way	100 x 100 board, 100 ants, 12 turns	getCritterCount in board class	The ant population should double about 5 times (a little less due to ants being occasionally unable to breed)	Performed as expected. 1581 ants, reasonably close to mean expectation.
Starving happens at the right times	19 columns, 20 rows, 9 steps, 95 ants, 1 doodlebugs	Db (standalone) function	Any 3 consecutive turns of a doodlebug not eating an ant should result in the doodlebug being removed from the board.	Performed as expected.
Doodlebug and ant population have inverse relationships	20 columns, 20 rows, 10 turns at a time (repeated) 100 ants, 5 doodlebugs,	n/a; simple visual inspection of the board across time.	Over time, board becoming dominated by the population of doodlebugs, then dominated by the population of ants, and this repeats.	Performed as expected.
Doodlebugs should eventually die out over long runs.	20 columns, 20 rows, 5300 turns, 100 ants, 5 doodlebugs	n/a; simple visual inspection of the board across time.	The doodlebugs died out.	Performed as expected.
Program does not crash due to being run for many turns	20 columns, 20 rows, 5300 turns, 100 ants, 5 doodlebugs	n/a	Program should not crash as there's no regular expansion of memory use, or arrays being filled to their maximum size	Performed as expected.
Make sure no unusual behavior on board sizes of 1 x 1	Board size of 1x1: 1 ant, 1 doodlebug, 0 critters. 10 turns for each of these.		Program should run with following all rules of larger boards.	Performed as expected.

REFLECTION

Design changes made

- At first we misunderstood the movement of the ants and doodlebugs. If they were going to move, we had coded the program so that they would only consider open spaces in the random choosing of the space to move to. So, we had to change the program to match the specs (the critters pick a random square first, then determine if it is available as secondary decision).
- We realized that the ants move before the doodlebugs so we just had to reverse the order of those two.
- The starving status of doodlebug was calculated and tested in the game class, with steps to check if there are ants to eat at the destination location. Bret improved it by addressing this in the doodlebug class, to keep the actions closely tied to the specified class without interacting with other classes it does not apply to.
- Initially, our movement function consisted of copying the ant that was in the current location, then deleting the original one. But, we changed this so that the pointer at the new location would point to the ant that already existed. We also changed the original pointer to then point to null.
- Our first implementation had separate classes for the actions of Ants and Doodlebugs. We ended up making more generic methods (getCritterSpaces, moveCritter), and instead pass a critterType as a parameter. This allowed to reduce duplicate code and simplify the implementation.
- It was tedious to keep outputting debugging messages, then commenting them out or deleting them when finished testing something. To save time, we created a function to display debugging messages. The function looked at a toggle variable to determine if it should output the debugging messages or not. Therefore we only had to set the var to true or false to turn all debugging message on or off.

Problems encountered and solutions

- When implementing starving, we needed to create an additional data member starvingDay and getStarvingDay function to track how many days the doodlebug has been starving for. Although this only applies to the doodlebug class, the compiler required it to be also included in the base class because our function getBoardCritter only returns a Critter object. However, it is considered to be against the object oriented design to include unrequired members to a class. Therefore, Bret found a way to fix it by static casting the Critter object to a Doodlebug object when calling the getStarvingDay function.
- When testing, it became tedious to count the number of ants and doodlebugs during each step to make sure breeding and starving were working correctly. We created a getCritterCount() method to count and display the number of ants and doodlebugs on the board during each step.

Lesson learned

- We saw first-hand how derived classes and base classes function. The starvation function helped us understand that a function that does not apply to a derived class should not be placed in the parent class.
- We all got great exposure and experience with Git, realized the importance of pull requests and code reviews, and discovered best practices for working on a project as a team.
- We learned the importance of tracking dynamically allocated memory, as well as using valgrind to identify memory leaks that existed in our programs.

WORK DISTRIBUTION:

Solomon Deutsch - Debugging, testing and fixing failures found, writing design document.
Eric Morgan - Setting up the skeleton/architecture of the program, some debugging
Steve Perreault - Major implementing and debugging, some documentation
Bret Swalberg - Major implementing and refactoring, some documentation
Chen Zou - Documentation, some implementing and debugging