1 Overview

In today's world, creating and testing simulations are becoming increasingly important, especially in fields like aerospace and nanotechnologies. Humans need to be able to create and verify that simulations are working correctly to implement objects in the real world.

1.1 Project Goals

In this project, we were asked to create a critter interpreter. This included reading in behaviors from critter description files and executing them correctly based on the guidelines given in the project assignment. Our methods were expected to take in critter files of various commands and lengths and run them in a specific order based on the surroundings. The goals of this project were also to learn more about creating and testing programs.

1.2 Personal Goals

Our personal goals were to create an interpreter that could correctly read in behaviors and execute them based on the parameters given in the assignment. In addition, we wanted to create an effective test harness that was not dependent on the simulation. Lastly, we both had personal goals of learning to verbalize our ideas and work together efficiently as a team, as we both have not had any experience with paired programming.

2 Solution Design

2.1 Assumptions

When we designed our loadCritter function, we made the assumption that we could access the critter file and there were no infinite loops inside of it. Furthermore, we assumed that the given code worked correctly.

2.2 Solution Design

Our solution has two interdependent functions: **executeCritter**, which performs critter actions based on the behaviors read in by the **loadCritter** function.

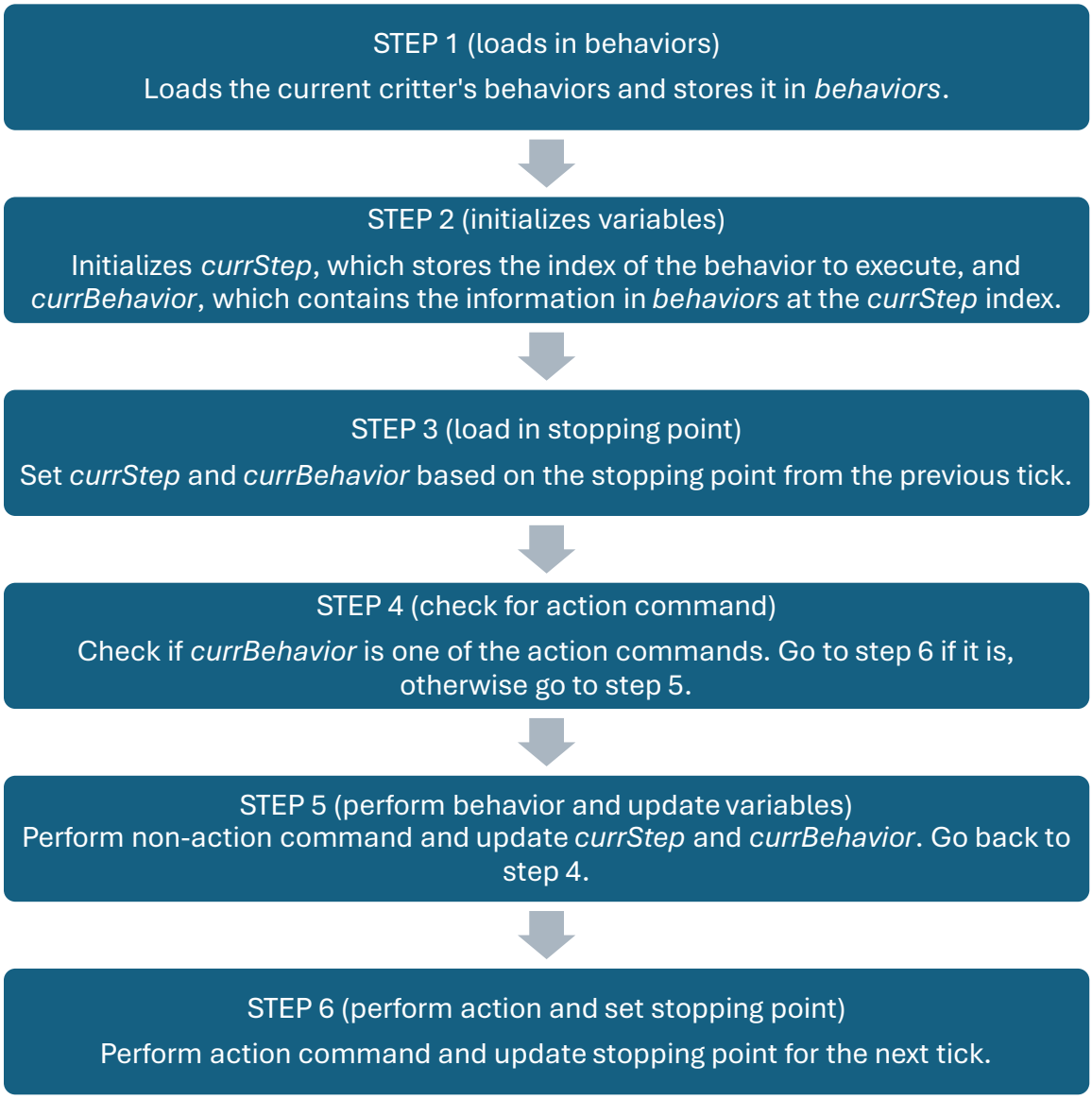
2.2.1 LoadCritter

LoadCritter reads in a behavior file and stores it in a nested ArrayList, with element *i* containing the information on the *i+1* line.

More specifically, the code starts by reading the first line of the critter file and storing it under the variable *name*. Then, the code iterates through the subsequent lines and stores them in the ArrayList *critterBehavior*, splitting each line by space to separate the command and the parameters. Finally, it returns a *critterSpecies* object with the information stored by the program.

2.2.2 Execute Critter

After the critter behavior has been loaded by **LoadCritter**, **ExecuteCritter** is called on each critter for the desired number of ticks. For the sake of clarity, we created a flow chart to graphically represent our coding structure.



Even though just using *currStep* was sufficient, we also used *currBehavior* for code readability.

2.2.3 Helper Functions

We created helper functions for the behaviors, each requiring 3 parameters: the current Critter, *currStep*, and *currBehavior*. The functions execute *currBehavior* and return an updated *currStep* value. We did this to help with code readability.

2.2.3.1 **isInt**

This function checks whether string parameters are integers. The code uses a try-catch block and returns a boolean of whether the string is able to be integer parsed. This function is a helper to the other functions listed below that validate the parameters in *currBehavior*.

2.2.3.2 **go**

The code checked the first character of the parameter for a +/-. If found, the code added/subtracted the proceeding integer to *currStep*. Otherwise, *currStep* was updated to the integer found in *currBehavior*. If the parameter was not a valid input, the program ended the turn and printed an error statement.

2.2.3.3 **ifrandom**

The code used the given *ifRandom* function to either execute our code or perform a no-op. If **ifRandom** returned true, then *currStep* was updated to the proceeding register or integer value. Otherwise, *currStep* was incremented by 1.

2.2.3.4 **ifhungry, ifstarving**

The code checked the hunger level of the critter using the given values. If the critter was hungry or starving, *currStep* was updated to the corresponding register or integer. Otherwise, the function moved to the next behavior line.

2.2.3.5 **ifempty, ifally, ifenemy, ifwall**

Using the given **getCellContent** method, our code checked the surrounding of the critter. If the respective condition was met, the code updated *currStep* to the register value or integer provided. Otherwise, it incremented *currStep* by 1.

2.2.3.6 **ifangle**

This function checked if the off angle bearing of the critter at b1 matched b2. If the check was true, *currStep* was updated to the respective register or integer value. Otherwise, the function incremented *currStep* by 1.

2.2.3.7 **iflt, ifeq, ifrt**

The code called the given **getReg** function to access the values in each register. If the inequality was true for the respective behavior, *currStep* was updated to the given register or integer. Otherwise, the code incremented *currStep* by 1.

2.2.3.8 **write, add, sub, inc, dec**

These methods did not require a helper function. The code used the given *setReg* method to update the desired register. Then, the code incremented *currStep* by 1.

3 Error Catching

3.1 Helper Functions

Each helper function checked to ensure that the parameter types matched that required by the command. In addition, the functions also verified that the number of parameters given were equal to the number required by the command.

3.2 Edge Case Testing

Below, we have listed the edge cases we tested and their results:

- Empty file: critter does not appear in the list of critters available for the simulation. Hence, empty files are not an issue.
- File with name and no behaviors, commands pointing to invalid indexes, behaviors with invalid command names, invalid integers/registers for inputs for behaviors: critter can be added to the simulation but it does not move.

4 Testing

We used JUnit Java testing to perform black box testing on **LoadSpecies** and white box testing on **ExecuteCritter**.

4.1 **LoadSpecies**

First, we tested **loadSpecies** by loading in a file from an invalid directory. Our program throws an IOException in this scenario, which we validated using a IOException assertion.

We also made our own CritterSpecies object to verify that **LoadSpecies** was working as intended. We took the example **food.cri** file and checked whether the returned behavior values matched that of the ones we manually created. Hence, if they matched, then we knew that our program was reading in the behaviors correctly.

4.2 **ExecuteCritter/Helper Functions**

The code in **ExecuteCritter** contains functions to test each helper function in the main **Interpreter** code. We created our own **Critter** interface for testing, along with an **EachCritter** object that implemented **Critter**. For each function, we created an **EachCritter** object and manually input behaviors. This allowed us to test that the code was incrementing *currStep* correctly based on the controlled environment that we set.

4.2.1 go

To test the **go** function, we created a new **EachCritter** object and added a **go** command with dummy parameter integer/register to its behaviors. Then, we used a test assertion to check whether the command was updating *currStep* to the line that we expected.

4.2.2 ifRandom

We tested this function by creating a new **EachCritter** object and adding an **ifRandom** command with dummy parameter integer/register to its behaviors. Then, we manually set **ifRandom** to true/false and checked to see if the *currStep* was being updated to the value we expected.

4.2.3 ifHungry, ifStarving

We created an **EachCritter** object and manually set its hunger to both true and false along with a dummy parameter integer/register. We then tested to see if the *currStep* was updated to our expected value.

4.2.4 ifEmpty, ifAlly, ifEnemy, ifWall

We created an **EachCritter** object and manually set the bearing to equal the value that we were testing for along with a dummy parameter integer/register.

In our **EachCritter** object, the **getCellContent** function just returns the bearing. Hence, by setting the bearing to 0, 1, 2, or 3, we could control our critter's surroundings easily and efficiently. For example, let's say we are testing **ifEmpty**. We would set the bearing of our **EachCritter** object to be 0. Hence, the **getCellContent** function would return 0, which would simulate when the critter saw an empty cell. Then, we would validate that *currStep* was jumping to the register or line given in the behavior that we manually put in.

4.2.5 ifAngle

We created an **EachCritter** object and manually set the bearing to 1 along with dummy parameters of integers/registers. Furthermore, the **getOffAngle** function for the

critter just returns the current bearing. This allows us to easily create an environment with a critter at *b1* with an equivalent bearing to the one at *b2*.

Then, we tested to see if our *currStep* was getting updated properly when the off angle matched and when it did not. After testing with integers, we tested the same conditions with register values to ensure our function worked with both types of parameters.

4.2.6 Iflt, ifgt, ifeq

We created an **EachCritter** object and set two register values using our **setReg** function. We then tested our function in all cases (e.g. Register 1 is greater than register 2, register 1 is less than register 2, etc.), asserting that the correct line jump would be executed. In addition, we tested the 3rd parameter with both integers and register values, ensuring the right line jumps were made.

4.2.7 Write, add, sub, inc, dec

We created an **EachCritter** object and added behaviors with the required number of parameters. We then performed each respective command and checked our output register value to ensure that the functions were setting each register to the value that was expected.

5 Discussion of Solution

5.1 Scope of Solution

loadSpecies can be useful in simulations with predefined behaviors for objects. Additionally, it can be helpful to organize data when given an input file.

executeCritter interprets the loaded behaviors and correctly executes them. This can be useful in simulations to run desired behaviors in patterns. For example, if we had an object like critter with specific behaviors, we could modify our functions to interpret the new language and run a simulation.

Overall, simulations are useful in the real world to predict future events like epidemics and test things that cannot be tested physically, like car collisions and structures over time.

5.2 Quality of Solution

Our interpreter works well for the critter language. It checks for valid behaviors while also executing commands in the proper order. In addition, we check for valid parameter types of both registers and integers. Finally, we created test code to ensure that our interpreter was incrementing *currStep* as explained in the assignment. Our solution can easily translate to other simulations if behavior files are structured in the same format and commands are modified to fit the needs of the object being simulated.

5.3 Problems Encountered

Throughout the project, we encountered many challenges. Below, we have listed them and the method we used to deal with them:

- Storing critter behaviors: we used *ArrayLists* because of its easy accessibility of specific behaviors and parameters.
- Invalid behavior commands or parameters: added checks in **executeCritter** helper functions to make sure that parameters matched those required by the command. We also made sure that all the commands were in the set of behaviors given in the assignment. We don't check this in **loadCritter** because there could be random commands that aren't valid but are never used. Hence, a critter shouldn't break in those cases.
- Testing methodology: we created our personal **Critter** interface that was implemented by a customized **EachCritter** object. This enabled us to easily manually control the environment of the critter. We also created a **CritterSpecies** object to test the **loadSpecies** method.

5.4 Interesting Results

The creation of our critter proved to be quite interesting. Logical decisions such as when to eat or infect were very intriguing to think about, and they encouraged us to discover different ways to optimize critter behavior. The results of many prototype critters were also quite confounding when tested. We found that rover was very well-made and often beat our prototypes, but also that there were strategies to optimize critter behavior. We quickly learned that critter commands could be implemented in a much more complex

manner than we had initially anticipated. For example, by simply adding wall checks and deleting wasteful moves in the rover critter, we were able to beat it.

5.5 Limitations

Our solution does not account for infinite logic loops in the given behavior file. Furthermore, it throws an IOException when it tries to read from a non-existent file due to an invalid directory. Finally, our critter stores behaviors in a nested ArrayList, which can be memory inefficient for large critter files. Rather than loading in all of each critter's behaviors and saving them, we could read in only the specific command we need from the critter file each time. Even though this increases the time complexity of our solution, it would save memory. We chose to save the time complexity rather than space under the assumption that the memory limit wouldn't be a problem. Below is a table giving a more explicit analysis of the time and space complexity of each method, where n is the size of a critter file and k is total size of all the critter files combined:

	Time	Space
Time efficient method	O(1)	O(k)
Memory Efficient method	O(n)	O(1)

5.6 Reflection

Overall, this project was a challenging task and induced deep thinking. We were forced to explain ideas to our partner before writing code, allowing us to catch most logical flaws before implementation. In addition, the testing strategy was difficult and encouraged collaboration to think of unique test harnesses. Overall, this project was a great introduction to paired programming and not only taught us to write clean, modular code, but also to collaborate effectively while programming with a partner.

6 Pair Programming Experience

Our paired programming experience was generally positive. While brainstorming solutions and testing methodologies, it was helpful to have a partner to talk to. In addition, verbalizing ideas allowed us to find logical flaws much more easily. Coding with a partner was also efficient because mistakes in syntax and logic were instantly found and corrected

by the driver. Finally, our report was significantly cleaner and easier to understand because there was feedback from a distinct perspective.

The biggest difference between paired programming and working alone was the necessity of communication. Every idea was thoroughly discussed before implementation, something that doesn't happen individually. On the other hand, a drawback of paired programming was the extra time spent implementing solutions. Explaining ideas to a partner slowed us down and resulted in a longer implementation time. Another difficulty we encountered was timing. Individual projects can be worked on at any time, while paired programming requires deliberate planning and set times for work.

6.1 Log of Time Spent on Project

Date:	**Activity:**
9/15	X drove for 3.5 hours
9/15	Y drove for 3.5 hours
9/18	X drove for 1 hour
9/18	Y drove for 1 hour
9/22	X drove for 3.5 hours
9/22	Y drove for 3.5 hours
9/23	X worked for 1 hour(report)
9/23	Y worked for 1 hour(report)
9/24	We brainstormed testing together for 1 hour
9/24	X drove for 1.5 hours
9/24	Y drove for 0.5 hours
9/24	Y worked for 2 hours(report)
9/24	X worked for 2 hours(report)
9/25	Y worked for 2 hours(report)
9/25	X worked for 2 hours(report)
9/26	X drove for 2 hours
9/26	Y worked for 2 hours(report)

7 Critter Creation

Our critter first checks the surrounding environment. If there is a wall or an ally in front of it, it randomly moves either left or right. Otherwise, if there is an empty space, it hops forward. Lastly, if there is an enemy in the space in front of our critter, it eats the

enemy if it is hungry. Otherwise, it simply infects it. This ensures that our critter only eats when necessary, allowing for more of our critter objects on the board at a given time. When we simulated this critter with many other test critters including rover, our critter beat the competition a high percentage of the time.