Critter Report
Niti Malwade and Didi Zhou
10/28/18

Overview of Assignment & Goals
This Critters assignment consisted of Critter objects which had certain functionalities that we were tasked with implementing into a simulation. The Interpreter class, which hosts all of the various methods that a Critter object can perform, implements the two abstract methods of the CritterInterpreter interface: executeCritter() and loadSpecies(). With both executeCritter() and loadSpecies(), the Critter should be able to read in instructions from a text file, and perform those behaviors in the simulation grid. This project relies on concepts of inheritance and hierarchy, and most importantly, abstraction. At first, we were definitely thrown off because we were not able to look inside the Critter methods, so we did not know exactly what was happening to the Critter. However, as we continued to discuss the project, we realized the power of abstraction - we could just assume that the methods worked and focus instead on how to implement those methods. Some goals we had for this project were to become familiar with Junit Testing and Mock Objects, and to evaluate the strengths and weaknesses of pair programming as a general concept, and more specifically, our effectiveness as pair programming partners.

Description of Solution Design
In order to perform as expected, the program makes a few assumptions. One assumption we made was that the code given to us works and would implement the methods we created as intended. Because of this assumption, many of our tests do not actually test the methods given to us, but rather, they test that we call them correctly. It is also assumes that the program is going to spend more time executing the critter than reading the text.

Our program also makes a lot of assumptions on how the actions given to us in the .cri file is made and formatted. It assumes the critter made by the user will be made in a logical way that will work. It assumes that there are no infinite loops made by the instructions for the critter being read in. Additionally, it assumes that the user will type in the actions in all lowercase. The program also assumes that the developer of the critter behavior will include and call at least one eat, hop, infect, or other action that will end the turn. The program also assumes that there will be a space between an action such as go and its parameter such as n for go. It also assumes that there will not be a space after the end of an action line.

One common algorithm that was used was to use .split(" ") especially on the behavior code to separate the action from the parameters that it came with and put each part of the action in an array. This makes it possible for us to compare the actions and use the line number values that follow them. We also use Integer.parseInt a lot since the array that the split creates is a String type so we have to parse them as ints to use them as desired.

Discussion of Completed Assignment
**Interpreter class**: The Interpreter class contains two methods inherited from CritterInterpreter - `executeCritter` and `loadSpecies` - which serve the role of taking in directions and assigning them to the Critter object. We also wrote a `go()` method in the Interpreter class which is called in the `executeCritter()` method for the majority of the if, go and state-manipulation instructions.

**loadSpecies()**: This method loads instructions from a file into an ArrayList so that `executeCritter` can read through all the instructions. `loadSpecies` takes in the filename as a parameter, and uses a Scanner to open and read through the file. In the Scanner, we used the File type to open the file of the specified name. First, we assigned the first line of the file to the name of the Critter, using the `scan.nextLine()` method. Then, the program calls a while loop and adds the next line of the file to the next index in the ArrayList `behaviors`. One assumption we had to make in order for `loadSpecies()` to work is that the instructions are in the correct format described in the instructions - name on the first line, and then one instruction per line after that with a blank line after the last instruction. In order to ensure that we do not have any empty indexes in the ArrayList, the while loop is only called when `scan.nextLine()` is true. After the program exits the while loop, it returns a new Critter with the parameters `name` and `behaviors`.

**executeCritter()**: In `executeCritter()`, we take in a Critter “c” as a parameter - the one created in `loadSpecies()` - and use this Critter to access the ArrayList `behaviorCode` which stores the list of behaviors loaded in from `loadSpecies()`. We then call a for loop which encompasses all of the actions or statements that a Critter can execute, including action statements such as eat or left, if statements such as `ifempty`, and `ifhungry`. The program traverses through `behaviorCode` and checks what the next instruction is and then goes to the corresponding code. To access each of these instructions, we use an if-statement and the “.equals()” operator to check which action the current instruction in `behaviorCode` matches to. For the instructions which require a parameter - either a line n, or a bearing b - we used the “.contains()” function rather than the “.equals()” function so that the if-statement can still evaluate to true even though it contains integers at the end of the instruction. In addition, when there is a bearing or line number after the instruction, we split the instruction at every space character and store each part in an ArrayList so that the instruction is in the 0th index, the bearing is in the 1st index, and the line number is in the 2nd index. This allows us to access and perform operations on the bearing or line number as an Integer in the code without having to deal with the name of the instruction, which is a String.

- **Action Statements:**
  For the first five action statements - left, right, eat, infect, and hop - `executeCritter()` simply calls the respective methods from the Critter class. For `go()`, we wrote our own method, also included in the Interpreter class, which is described below.

- **If Statements:**
  For `ifRandom()`, we also called the `ifRandom()` method from the Critter class. `ifhungry` and `ifstarving` were a little more complex - we included an if statement which checks if the hunger level of the Critter is either hungry or starving. If this is true, then the program calls our `go` method. The same concept was applied for `ifempty`, `ifally`, `ifenemy`, and `ifwall`, except instead of
calling the getHungerLevel() method, getCellContent() was called to check what was in the
adjacent cell at the specified bearing. If the if-statement was true, we called go(). For all the
static variables of the Critter and HungerLevel classes (SATISFIED, STARVING, HUNGRY,
ENEMY, ALLY, EMPTY, WALL, BAD), we learned that we must call the Critter class or the
Critter.HungerLevel class in order to access these variables. For ifangle, the same structure was
used, except getOffAngle() was called to find the difference between two Critters’ bearings. At
first, we did not include any “else” statements, but as we tested our code, we realized that we
needed an else clause with a continue statement in order for the for loop to continue evaluating
the rest of the if-statements.

- Registers:
For the methods involving changing the values of the registers - write, add, sub, inc, and dec -
we did not use if-statements. Instead, we called setReg() from the Critter class and performed
the required arithmetic inside the setReg() parameters. Lastly, for comparing the registers (iflt,
ifgt, and ifeq), we used getReg() to compare the values of the two specified registers, and call
go(), if the statement evaluated to true.

go(): This method takes in a String “num”, integer “i”, and Critter “c” as parameters and returns
an integer. “num” is the current instruction in the behaviorCode ArrayList, the integer is the
current index of behaviorCode, and the Critter object is the one called in executeCritter(). First,
our method checks for the three possible types of parameters of go: +/- line number, line
number, or a register. We do this by creating three if-statements and checking the character at
the index 0 of “num”. If there is a +/-, we increment “i” by the specified amount (we learned that
you can parse +/- to an integer as well and it just serves to define the integer as a positive or
negative number). If there is just a line number, we change “i” to the specified number, and if
there is a register, we set “i” equal to the value of the specified register. Then, we exit the
if-statement and return “i-2”. We return “i-2” because when the go() method is called in
executeCritter(), the method needs to know which index to jump to on the next iteration of the
for loop. We subtract 2 from “i” because the instructions starts at index 1, so we subtract 1 to
start the index of the ArrayList at 0. Then we subtract 1 again because the for loop will naturally
increment i to one more than the value that we just set i to. For example, if we set i to 4 in go(),
then the index of behaviorCode should be at 3 instead of 4, but also when the for-loop
reiterates, it will increment 4 to 5. Therefore, we subtract 2 to get an index of 3.

About the Solution:
Our interpreter is very simple as it contains elements such as if statements and registers
but is lacking in more complex elements like loops and higher level data structures. In terms of
feature within our interpreter, the scope of ours was pretty weak. Our program was also not
extremely robust as it is very easy to create an infinite loop inside the code for an individual
critter and the critter would not run. This would happen when the turn just never ended and only
if statements were repeatedly called.

One design problem our interpreter had was running into many off-by-one errors due to
a difference in how our arraylist storing the actions of the critter were designed and how the line
numbers in the actual code were numbered. This is described in more detail in the “Problems Encountered” section of the report. One good design strategy was the use of abstraction to make a go method in our code in order to change the line number that the code was on. Because many of the functions needed to jump to different lines, having one go method that was called for each action was much more efficient than redundantly using the same code multiple times in the program.

If given more time, we would have tried to fix some of these issues and make the code more robust. We would have tried to add more features like different kinds of loops into the critter language and we would have checked for infinite loops in the program.

**About the Critter:**

In order to create a good critter, we ran some simulations and saw that, overall, rover performed the best out of the given test critters. Based on this observation, we decided to approach the creation of our own critter through using the logic of the code behind rover and making adjustments. We looked for the flaws with rover and fix those. The main issue we fixed with rover was that it would waste turns randomly turning left or right when the square by it wasn’t empty. This would give the rover a significant disadvantage over critters who logically decided when to turn. In order to fix this, our critter checks where there is a wall or ally nearby and based off that, will turn either left or right, depending on which bearing there is a wall or ally.

Our critter, Malwade, hops in a straight line until it is blocked, eating enemies it sees if it is hungry and infecting them otherwise. When it is blocked, it checks for where there are walls or ally around it and, based off of the bearing of it, will turn the least needed number of turns to become unblocked. This gives Malwade an advantage over rover and allows it to beat rover and the rest of the test critters.

**Problems Encountered:**

Throughout the course of this assignment, we ran into many problems that ranged from off by one errors and syntax errors to larger errors that changed the behaviors of the critters. Some of the memorable errors are described below.

A problem we had was with off-by-one errors. This was caused by the line numbers in the critter’s code starting from one but the arraylist they were stored in’s indexes starting at 0. This problem was solved by adding or subtracting one to the position in the for loop every time we needed to change between the line numbers and the arraylist index. This problem let to a lot of confusion and time spent trying to decipher what line code is leading to and to a lot of the errors in our code. One way we could have solved this problem was to start our arraylist with an empty element to make the numbers match up in the arraylist and in the lines.

Another problem we encountered was a result of the previous overarching problem, but we would sometimes try to access negative lines in the arraylist because of when the for loop would increment the value we would have to subtract two from the i value. Once to compensate for the incrementing of the for loop and another to actually subtract one. Sometimes, this would lead to i being negative. In order to fix this, we created an if statement at the beginning of the code that would check if i was less than zero. If it was, we would set i equal to zero.
At the beginning, we ran into problems with our program randomly freezing when we tried to start it. We then realized that this was due to an infinite loop being created somewhere. We looped through our code but couldn’t find anywhere that may cause this infinite loop and were pretty confused. While looking for the bug, we found other small bugs such as off by one errors and syntax errors, but we could not figure out why our program kept freezing. We then looked in our code for the critter and realized that while adding new lines of actions, we forgot to go back and change the line parameters on go methods or on if statements. To prevent this from happening again and to save us from counting through the code line by line, we put our critter instructions in eclipse and used the line numbers on the side to count for us. This prevented issues with line numbers in the critter code from occuring after that.

**Interesting Results:**

I found it interesting to see how the critters acted and how their actions changed due to the actions of other critters. I thought it was interesting how significantly our critters would slow down at a certain point once all the food was eaten and also how significantly the number of overall critters would decrease at the end compared to the number of critters at the start of the simulation since they would get eaten.

Some interesting Java skills we learned included how to use Junit testing. Prior to this project, neither of us had used Junit testing before and so building the test harness was a learning experience for both of us. We watched some Youtube videos and online tutorials in order to figure out how to implement the Junit testing. Additionally, we needed to create a mock object to test our code. We had to override some of the methods in the critter class that we couldn’t access in order to make sure we were calling it right. However, now that we have learned how to implement Junit testing, it can be applied to testing other programs and is very useful.

**Good Karma:**

It was very interesting to observe the behaviors of the critters and how these behaviors affected other critters. The critters that placed more emphasis on infecting than on eating tended to do well at the beginning but then quickly slowed down and decreased in number towards the end. When the program ran for a long time and ran out of food, I noticed that the number of critters at the end would decrease significantly from the number of critters that were put into the simulation because the critters would eat each other. Additionally, because some of the critters use the presence of another critter to determine their moves, these critters tend to move around more and turn more when there are more critters of any kind in the simulation.

For good karma, we also designed many different critters to observe how they performed. A description of them are described below:

- **Cluster** - Cluster hops around and randomly turns until it can find an ally. Once it finds an ally, it stops and they start spinning together, infecting anyone that they come in contact with.
- **Passive** - Passive sits in place and waits until an enemy is in front of it. When an enemy comes in front of it, it will infect the enemy and then turn left.
• Random - Random will randomly hop and turn left and right when there is not an enemy in front of it. If there is an enemy, it will check if the critter is starving and then eat, but if it is not starving, it will infect the enemy.
• MalwadeEats - MalwadeEats is the same as Malwade, except instead of infecting other critters, it will only eat them.
• MalwadeStarving - MalwadeStarving is the same as Malwade, except it will only eat other critters if it is starving.

Some other ideas for critters that we had but did not have time to implement included a critter that would find ally critters and then cluster together and move around infecting other critters as a big cluster. We also wanted to create a critter that went found a corner and just clustered there like a flytrap in order to reduce the number of sides that an enemy could infect or eat it from.

Software Test Methodology

Black Box and White Box Testing

In order to test our code, black box, white box testing, Junit testing, and mock objects were used. At first, most of the testing was done using black box testing as we would run the premade Critters in the Critter.jar file and evaluate if they were performing the expected actions. This testing method was not very useful to us however, because our program would sometimes freeze or crash, and we could not identify which methods were causing the program to break. In particular, we reached a point where all of the premade Critters would run as expected except Rover - which caused the simulation to freeze. To debug, we created our own test critter and called each method individually to identify which one was causing the error. From there, we utilized white box testing to check each segment of our program and were able to narrow down the source of the error to ifempty. We realized we were calling an infinite loop, and were not continuing out of the if-statement. This testing method was very useful to us, because it helped us distinguish the source of our error.

Junit Testing and Mock Objects: executeCritter()

After that, all of the premade Critters ran accurately in the simulation jar, but we wanted to do more detailed testing to ensure that the correct methods were actually being called. In order to do this, we used Junit Testing. We created a Mock Critter class which implemented the Critter class and served as a mock object where we could manipulate each method to do what we wanted. In general, we created multiple integer objects which serve as counters, and then we overwrote the methods to increment the counter. Then, in our Junit Testing class, we called assertEquals and compared the expected value of the counter to the actual value of the counter using getter methods.

• Eat, hop, left, right, infect, ifrandom, ifangle

We used this method for the eat(), hop(), left(), right(), and infect() methods. We felt this was a sufficient way to test our code because we were not necessarily interested in if the actual code in the Critter class was correct - we can already assume that the code you supplied us
with will work - we just wanted to make sure we were calling the correct methods at different points in the program. An increment in the counter indicated to us that the method had been called.

Similar methods were used to test if ifrandom() and ifangle() are called. For ifrandom(), we created a boolean variable in the MockCritter class and set it to false; in the ifrandom() method, we set it to true and then checked in the ExecuteCritterTest() class if ifrandom() returned true. For ifangle(), we set getOffAngle() to return 0 and then checked if ifangle() returned 0. We believe that even though our testing methods oversimplify the complexities of the program and leave a majority of the code to abstraction, it serves the purpose to check if the correct methods are being called and arithmetic is being performed.

- ifempty, ifwall, ifally, ifenemy
  Part of what makes this program difficult to test are the many aspects of randomization within it - there are numerous Critters moving around at different paces, and in different directions, so it is hard to ensure that the code is actually being called. To test that the code works without the randomization, we replaced the randomized parts of the code with hard-programmed values. For ifempty(), ifwall(), ifally(), and ifenemy(), we modified the getCellContent() method to essentially create our own mock cell in the grid. We assigned an enemy, ally, wall, and empty cell to different bearings around the center cell using the numerical values of each variable (e.g. enemy = 2). This way, when we called ifempty(), for example, we knew that there should be an empty cell at a bearing of 0, so we could check if getCellContent() at a bearing of 0 evaluated to “empty”. If the statement evaluated to true, we knew that our if-statement for ifempty() in the Interpreter class was accurately evaluating the state of an adjacent cell. To test that the cell state “BAD” would be outputted when an invalid bearing was entered, we called getCellContent() of a bearing of 2 and evaluated if this statement was equal to the Critter.BAD variable.

- ifhungry, ifstarving
  ifhungry() and ifstarving() were a little harder to test because the getHungerLevel() method in the Critter class does not take in any parameters - therefore, we could not modify the hunger level in any way. Furthermore, HUNGRY, SATISFIED, and STARVING do not have numerical equivalents like ALLY, ENEMY, etc. do, so we could not directly compare them to an integer value either as we did previously. After talking to Michael about this issue in TA office hours, we decided to create an integer variable in the MockCritter class “hungerValue” which would be incremented every time the hop() method was called. Then, in getHungerLevel(), we created three if-statements where 1, 2, and 3, would be equated to SATISFIED, HUNGRY, and STARVING, respectively. This made it easy to check in the Junit Test to check, because we could directly control the hunger state by calling hop() various amounts of times and then check if it was equivalent to getHungerLevel(). Although this was not the cleanest way to test - because it relies on other code which we are also testing - it was the most straightforward way to check if the hunger levels are evaluating correctly. Overall, this strategy is not too problematic because the hop() method is pretty simple, so it was very clear to us how each variable was being modified.
- **Registers**
  For the methods involving the registers, we created an integer variable “register” and had `getReg()` return “register” and `setReg()` set “arg1” (the second parameter) equal to “register”. Then, for `write()`, `add()`, `sub()`, `inc()`, and `dec()`, we made the respective arithmetic modifications to `setReg()`. For `iflt()`, `ifgt()`, and `ifeq()`, we used `getReg()` to compare the values of 2 different registers.

- **Go**
  Lastly, to test the `go()` method which we wrote in Interpreter, we simply copied over our code to the `ExecuteCritterTest` class and replaced variables such as “num” (a String value from the `behaviorCode` ArrayList) and `i` (index of `behaviorCode`) with hard-coded String and integer values. Then we checked if the actual output matched the expected based on our inputs. We created 3 different versions of the `go()` test - one for an increment/decrement, one for a line number, and a last one for a register input.

**Unit Testing: loadSpecies()**
For the `loadSpecies()` method in Interpreter, we did not use Junit Testing, because we did not feel it was necessary. Instead, we simply inserted a print line in the while loop, so we could see every item loaded into the ArrayList, and compare the results with the instructions in the .cri file.

.Unit testing was also used by inserting print statements to the code not only in `loadSpecies()`, but also in the action methods, to check that new code line was set correctly, to check that we split the input String correctly at a space, to check that `go()` was returning the correct value of “i”, and to check if the `add()`, `sub()`, `inc()`, and `dec()` methods were actually changing the register values. This was especially useful to see where exactly mistakes were created since a value could be tracked at each point where it may be changed and see where it is not as expected.

**Further Testing** could have been used to increase confidence in the program. In order to fully ensure that our Critters were performing the correct actions, we could have created our own .jar file/simulation so we could test the critters in a more efficient, controlled environment. For instance, we could have made the grid smaller, which would have helped us identify strengths and weaknesses of different Critters and features on a smaller scale before expanding to a larger, more randomized environment. We could also have allowed user input so that it would be easier to test specific actions - where the user can just input “hop()” for example and see if the Critter moves forward one space, rather than having to edit the .cri file every time.

**Pair Programming**

<table>
<thead>
<tr>
<th>Date</th>
<th>Driving</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/17</td>
<td>Niti: 1 hr</td>
<td>n/a</td>
</tr>
<tr>
<td>Date</td>
<td>Niti: Time</td>
<td>Didi: Time</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>9/18</td>
<td>30 min</td>
<td>1 hr</td>
</tr>
<tr>
<td>9/19</td>
<td>2 hrs</td>
<td>2 hrs</td>
</tr>
<tr>
<td>9/22</td>
<td>45 min</td>
<td>1 hr 15 min</td>
</tr>
<tr>
<td>9/23</td>
<td>1 hr</td>
<td>1 hr</td>
</tr>
<tr>
<td>9/24</td>
<td>1 hr 30 min</td>
<td>30 min</td>
</tr>
<tr>
<td>9/25</td>
<td>1 hr 15 min</td>
<td>1 hr 15 min</td>
</tr>
<tr>
<td>9/26</td>
<td>1 hr 30 min</td>
<td>1 hr 30 min</td>
</tr>
<tr>
<td>9/27</td>
<td>45 min</td>
<td>15 min</td>
</tr>
<tr>
<td>9/28</td>
<td>30 min</td>
<td>30 min</td>
</tr>
</tbody>
</table>

For us, pair programming was very effective in producing correct code as it allowed our errors to be caught right when they were made rather than when compiling. This saves us time from having to scour the code for bugs when they appear at compile time. However, when we do have to look for errors, it is usually twice as fast because there are two sets of eyes on the code, looking for it. We pair programmed by first talking through exactly what we were going to do before anyone began to drive. This allowed both of us to fully understand what we were doing before just going for it. Pair programming is different from working alone as it requires both partners to coordinate when they will meet up and when they will code. When working alone, we could program at any time when we were free, however, with pair programming, we set aside time to work on the program. This kept us from procrastinating the project but also was difficult due to how busy we were. Sometimes we couldn’t find time to meet up until late at night which lead to both of us being easily distracted and tired while coding.