1 Introduction and goals

This assignment explores developing an interpreter used in the simulation of critters whose possible actions are given by a primitive programming language. In addition to implementing the interpreter, we are tasked with developing an effective test strategy involving a test harness and writing the code for battle-ready critters.

This assignment differs significantly from those before it in that it contains three levels of abstraction: a provided simulator to process players’ turns and display output, an interpreter to translate critter code into instructions for the simulator, and the critter code itself.

Other than successfully completing the interpreter, we sought to develop an effective test harness and to create interesting critters for use in the CritterFest that utilize group behavior and novel strategies such as the eat action. We also wanted to learn more about automated testing using JUnit.

2 Solution design

2.1 Basic design patterns and abstractions

Unless a complex compiler is developed, which was outside the scope of our goals for this assignment, it is not possible to determine when a critter’s turn should end without manually iterating through the program flow specified in the critter code. The interpreter must thus loop through the instructions until an instruction that ends its turn is encountered. We accomplish this using a switch-case statement to map the root name of an instruction to the behavior that must occur when it is called. At the end of each case in the switch-case statement, if the critter’s turn should end, return is called; otherwise, break is called.

The critter interpreter serves as a layer of abstraction that allows a simulator to execute critter code. The simulator relies on an interface that contains two functions, loadSpecies and executeCritter. Our specific implementation of these methods should not influence the function of the simulator or the specification of the critter code.

2.2 Interface implementation

2.2.1 loadSpecies

The loadSpecies method is invoked by the simulator to load the instructions stored in a critter file. The file is read until a blank line is encountered, populating a list of Instructions. Before each line of critter code is added to the list of instructions, its root name and parameter lists are checked for syntax errors. If an error is encountered, loadSpecies returns null and the critter file
is not used in the simulation. If no errors are found, a new CritterSpecies, populated with the
name of the species (a String), and its behavior code (an ArrayList<Instruction>), is returned.

2.2.2 executeCritter
The executeCritter method is invoked by the simulator to process the actions the critter performs
on one turn. The method executes lines of critter code, incrementing the critter’s internal current
code line as necessary until it encounters an instruction that causes its turn to end. To do this,
it passes the root name of the current command to a switch-case statement, making calls to the
Critter interface methods using the parameters as needed.

2.3 Instruction class
To simplify the internal storage of instructions in the CritterSpecies class, we store each line
of critter code in a class called Instruction. Each instance of this class stores the root name
of the instruction and a list of parameters. The parameters are stored as Strings to account for
irregularities such as register addresses being prefixed by the character ‘r’ and relative jumps being
prefixed by ‘+’ or ‘−’.

2.4 Helper methods (isNumericString, isRegister, isBearing)
In order to simplify validation of input in loadSpecies, we created several helper methods that
take Strings and return true if they represent valid numeric values, register addresss, or bearings.
A valid numeric string is a number optionally prefixed by a ‘+’ or ‘−’, a valid register address is
a number between 1 and 10 (inclusive) that is prefixed with ‘r,’ and a valid bearing is an integer
under 360 that is divisible by 45.

2.5 Extensions
We did not have the time do develop a comprehensive critter code compiler for this assignment, but
made many interesting observations on the behaviors of critters in general and developed several
critters that fare well in match-ups with other critters, discussed in sections 3.4 and 5.

3 Discussion

3.1 Assumptions and limitations
In our implementation for reading critter files, we assume that the author of the critter code provides
one critter code instruction and its parameter in a single line of the file. We also assume that the
end of input is properly formed; namely, that there is no blank line midway through the input. By
the project specification, the input is ended by a blank line, so whenever our loadSpecies method
encounters a blank line, reading input stops. There is no way to anticipate and prevent this.

We assume that a simulator using our implementation of the CritterInterpreter interface
only uses the loadSpecies method to load critter code from data files. Since our error handling
is in this method, a simulator that manually loads critter code from elsewhere could successfully
load code with syntax errors.

We are limited in that we cannot anticipate bugs in the simulator that is used with our inter-
preter. Even if the interpreter makes the correct method calls in the Critter interface, we cannot
be sure of simulator implementation details such as the order that critters of opposing species take their turns.

Another important limitation we face is that the Critter interface’s action methods (hop, eat, etc.) simply tell the simulator the critter’s intended path of action, but do not actually execute the action at that moment. Thus, in the case where two critters are facing each other that both intend to perform the infect action, the determination of which critter acts first is made by the simulator, not the interpreter. This somewhat limits the strategic considerations we can make in designing our critters.

3.2 Scope and style

Based on the structure of the interpreter interface, our solution should be applicable to any simulator that correctly uses the interface and implements the underlying critter handling.

Adhering to typical documentation standards, we have provided Javadoc-style documentation for classes and methods, as well as comments for specific functions that may not be clear based on the code itself.

3.3 Edge cases

3.3.1 Infinite loops

An interesting edge case we encountered is that when the critter code contains an infinite loop of the form

1  go  2
2  go  1

To prevent the simulator from being caught in an infinite loop, we created an upper bound of 1,000 non-turn-ending operations that can be performed by a critter in a turn. If this limit is exceeded, the critter terminates its turn without performing an action.

Another case we thought about but decided to allow is the possibility of a register value exceeding the maximum integer value in Java. Overflowing in registers occurs as would be expected with normal Java ints.

3.3.2 Malformed input

Most other edge cases are handled by our loadSpecies method, which returns null if any syntax errors are present in the critter code while reading files. It also returns null if the critter file contains no code or is empty.

3.4 Insights and interesting results

3.4.1 Insights

The fact that critters can only detect those critters directly adjacent to them and can typically only execute one action per turn somewhat limits their behavior when large groups of critters are present. Our implementation of locating enemies has a preference for those in front of the critter and may ignore those behind it, causing it to stay in dangerous situations and be attacked.

Some earlier revisions of our critter for CritterFest (see Section 5) were more aggressive and never moved to safety if they could infect another critter. Our current version is more self-aware and runs away if there is a significant chance it could be attacked. If more information were available
to the critter, such as whether it was moving first, it could make more informed decisions about
whether to infect or run away.

3.4.2 Interesting results

We observed several intriguing results while running simulations, including inconclusive results in
critter match-ups and scenarios in which critter populations became stable.

When running simulations with the provided critter species, we noticed that the rover species
tended to overwhelm flytrap species when matched up against them, even when significantly out-
numbered. However, in rare cases, if the flytraps could establish a stable clump, it would slowly
grow to overwhelm the rovers.

This case gave us the idea to implement new critters that combine clumping with more mobile
movement. When multiple critter species that employ movement with clumping techniques were
used, situations often emerged when critters would establish stable colonies and not have the
opportunity to infect or eat one another. In one case, two original rovers were able to survive
indefinitely against several hundred of our modified rovers because a cycle of infections emerged.

3.5 Personal reflection

This assignment allowed us to get a much more practical exposure to the design considerations
and consequences of abstraction techniques in software development. The separation of concerns
that abstraction enables allows us to write code that can be used with environments as diverse as
simulations and testing harnesses without needing modifications of our interpreter implementation.

Writing new critter species using the primitive critter programming language was an exercise in
how programming would feel without the numerous scaffolds and conveniences afforded by modern
and object-oriented programming languages. Even basic structures such as terminating loops are
difficult to implement, and simple changes to the codebase often require refactoring the code that
has already been written to account for insertions and deletions of lines of critter code.

4 Software test methodology

While it is tempting to use the provided critter simulation software to test our code, doing this
would constitute an ineffective test strategy because we cannot see or control the implementation of
the provided simulation software. This would limit us largely to black box testing without knowing
the internal details of how critter commands are processed. We would have no guarantee that the
interpreter actually makes the correct calls to the simulator.

4.1 Black box testing

In the very early stages of development, we used black box testing with the supplied simulator to
verify that critters behave generally as we expect them to (that the flytrap rotates, that the rover
roams, etc.).

4.2 Test harness (unit testing with JUnit)

To overcome this pitfall, we developed our own custom test harness to use in place of the provided
simulation software. We created two classes, TestCritter and TestCases.
- TestCritter implements the Critter interface and its methods; however, it does not act as a normal critter in the simulation would. TestCritter’s methods either return dummy values that are useful for testing or set the class’s String member to the name of the most recently called command. This allows us to control all variables in the interpreter without relying on the provided simulator.

- TestCases uses the JUnit library to test a variety of branches that could occur in a specific critter’s code. These tests all follow the same structure: each manually sets the critter’s code line to one containing the command in question in the test file, setting any required registers, and calling executeCritter. For commands such as go where the code line must be changed, or where we assume conditionals would return true, our expected behavior is that the test code should jump to line 1. Line 1 of our test file is a hop command, so these tests assert that the current code line is 2 and the last action performed is hop after the jump is performed. In the other, more simple tests, we only consider whether the String containing the previous action is updated and the code line increments. The only exception is the case of go commands using the syntax for relative jumps in the positive direction. We cannot return to line 1, so we specify a line farther down in the file, line 12, for convenience.

4.3 Strengths and weaknesses

While our test harness can establish that our interpreter makes the correct calls to the underlying Critter interface, it cannot guarantee the correctness of the simulator itself. Errors in the simulator (such as improper handling of the eat method) depend on the simulator used, not the interpreter.

Our testing is also limited by the fact that we cannot anticipate all possible paths through our code since it relies on arbitrary-length critter files that involve their own branching statements.

5 Created critters and strategic considerations

5.1 CritterFest critter

The critter we plan to submit for CritterFest is derived from the provided rover species. In a situation when it needs to eat or infect another critter, it calls eat if it does not already have a speed boost from eating. If it has already eaten and has a boost, it calls infect.

In a typical turn, our critter checks for enemies in adjacent cells. If there is an enemy directly in front of it, it eats or infects it, as described above. If an enemy is at an angle of 45° or 90°, it will turn toward it. If a wall is in front of it, it will randomly turn left or right.

We have some future development ideas to improve our critter before CritterFest. Since our critter turns randomly when it encounters a wall, it may repeatedly turn left, then right, causing it to get stuck. We could modify the critter so that it chooses one direction and sticks with it.

5.2 Other critters and strategies

We also experimented with some other critters, described below:

- LedgeCamper, a critter that behaves like the standard rover until it is adjacent to a wall, in which case it stops hopping, turns away from the wall, and turns toward and infects adjacent enemies.

- Clumper, a critter that behaves similarly to the LedgeCamper, but stops moving when it is adjacent to a friendly critter rather than a wall.
6 Pair programming work log and reflection

6.1 Work log

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 September 2017</td>
<td>• 30 minutes: Marcus driving, Ben navigating</td>
</tr>
<tr>
<td></td>
<td>• 1 hour 30 minutes: Ben driving, Marcus navigating</td>
</tr>
<tr>
<td>23 September 2017</td>
<td>• 1 hour 30 minutes: Marcus driving, Ben navigating</td>
</tr>
<tr>
<td>24 September 2017</td>
<td>• 1 hour: Marcus driving, Ben navigating</td>
</tr>
<tr>
<td></td>
<td>• 1 hour: Ben driving, Marcus navigating</td>
</tr>
<tr>
<td></td>
<td>• 1 hour: critter designing (not as a pair)</td>
</tr>
<tr>
<td>25 September 2017</td>
<td>• 2 hours: critters designing (not as a pair)</td>
</tr>
<tr>
<td>27 September 2017</td>
<td>• 4 hours: Marcus drafting report and Ben adding error handling,</td>
</tr>
<tr>
<td></td>
<td>Marcus implementing test harness and Ben refining report</td>
</tr>
<tr>
<td>28 September 2017</td>
<td>• 4 hours: Marcus finishing test harness and Ben finishing report</td>
</tr>
</tbody>
</table>

6.2 Reflection

We found that the pair programming experience encouraged a broader discussion of solution strategies that each of us would be unlikely to have thought of on our own. Working as a pair certainly cut down on the amount of debugging because we could have a dialogue about solution design and catch minor syntax errors as they were introduced. We may have been limited in that both of us have similar skill levels and were not able to learn a considerable amount from each other.

In future pair programming situations, it would be useful to use a more streamlined method for tracking code changes, such as a private GitHub repository.

References

We referred to Tres and Aditya for general ideas on how to implement a test harness.