Contents

1 Abstract

1.1 Overview

We implement `Interpreter.java`, which has two primary functions:

1. `Interpreter` can parse in an input text file and use the parsed contents to initialize a ”species” of critters, whose instructions are detailed in and parsed from the input file.
2. `Interpreter` gives the critters the ability to execute the parsed instructions for the critters in any critter species.

1.2 Goals

We wanted to familiarize ourselves with pair programming, including figuring out scheduling, distributing the workload, and utilizing git to collaborate across devices.

We also wanted to get practice implementing OOP, since we have little experience with projects involving many classes, so using OOP to complete `Critter` gives us a chance to practice OOP methods learned in class, simplifying the code significantly.

Finally, we wanted to further practice automated testing and learn new testing techniques by trying new testing packages and creating a comprehensive set of testing classes.

2 Solution Design

2.1 Class Structure & Underlying Data Structure

We create a class umbrella that stems from the `Command` abstract class, which provides basic input validation methods and abstract method structure. Since all other commands extend from this class, we expect all commands to have the following attributes & behaviors:

- `args`, an `ArrayList` which stores each argument for the command.
- `minNumArgs` and `maxNumArgs`, which set bounds on the number of arguments each individual command is allowed.
- `processToken()`, which processes the arguments, unless an error occurs while doing so.
- `executeCommand()`, which executes the class's command with the specific arguments preprocessed, and returns the next line of code that should be run (either the next line, or the line number corresponding to a jump)

2.1.1 Naming

Depending on the argument structure of each command, we've created classes which parse arguments. For example, the commands `ifEnemy` and `ifWall` both take a bearing and a line jump as arguments. The `Command` class for both `ifEnemy` and `ifWall` will therefore both extend a class names `BearingJumpArgumentCommand`, which handles the argument processing for both commands.

2.2 Overview Flow

When we read in a file, we first validate the parameters the file contains. If the file can be parsed properly, we pass the series of `Command` objects into the CritterSpecies. If parsing the instructions fails for whatever reason, we return `null`.

`executeCritter()` will retrieve the command list and call `Command.executeCommand()` for each command. The critter will then execute commands in order until it reaches a line out of the bounds of its `commandList`.

2.3 Processing Commands

2.3.1 Reading in the File

Before we even read in the file in `loadSpecies(String filename)`, we perform various checks on the file:

1. We check that `String filename` is non-null
2. We check that the file at `filename` exists and is readable

2.3.2 Reading in the lines of the file

For each line in the file, we use the `StringTokenizer` class to split the line by white-space. We parse the first token as the command name, and any remaining token afterwards as arguments.

From here, we return `null` if anything breaks for any reason. This includes the following:

- First token is unrecognizable/not a valid command token
- Number of following arguments is incorrect for the command
- Argument formats are incorrect.
- We reach a blank/only white-space line in the original file

2.3.3 Processing Arguments

For every command, we process arguments in the same fashion:

1. Check that there are the correct number of arguments.
2. Check that each argument is the correct format:
    - For any token, check if the numeric part is unsigned and parseable as an integer
    - If expected token is a bearing, check that it is one of the possible bearing values.
    - If expected token is a register, check that token begins with an 'r' and contains an unsigned integer between 1 and 10.
    - If expected token is a line jump, check that token begins with an 'r', '+', '-', or a numeric character.
3. If arguments are valid/can be parsed, parse them and add them to the `ArrayList<Integer> args`, keeping only the integer part.

- If the argument is a line jump, also store whether or not the integer is a relative, register, or absolute jump.

4. Repeat until all tokens are parsed as arguments
   - If any of the checks fail, set `ArrayList<Integer> args` to null.

If, by the end of argument processing, we find that `ArrayList<Integer> args` is null, then we know there was an error in argument processing, so we return null out of the outer `Interpreter`. Otherwise, we return the `CritterSpecies` with a `commandList` containing `Command`s intialized with the arguments as parsed through the file.

2.4 Executing Commands

Beginning at the first `Command` in `commandList`, we run `Command.executeCommand()`. If the command's execution involves a line jump, then we perform the next `Command` at the targeted line. Otherwise, we perform the `Command` at the following line.

2.5 Design Decisions

1. If the file ends after the last line of critter code, we still consider this to be a valid file, with the last critter instruction in the file considered to be the last command.
2. All white-space, within or after the critter instructions, is ignored. For example, have multiple spaces between command arguments is still considered valid. Furthermore, if we reach a line with only white-space, we consider this to be a blank line, and treat that line as the end of the critter instructions.-

3 Reflection

3.1 Assumptions

- The methods in the Critter Interface work as intended.
- Any Critter objects passed in through methods are immutable.
- The input file remains immutable while the `CritterSpecies` is loaded.
- The file has less than `Integer.MAX_VALUE` lines.
- No integers have leading zeros.

3.2 Solution Scope

`Interpreter` will generate a `CritterSpecies` with the commands given in an input file and allow for the execution of these commands as given by the file, within the rules of the commands. Our `Interpreter` implementation should work with any valid critter code file, as it implements all functionality related to any critter behavior. However, the scope is limited because the program can only work with critter code files, and is not able to interpret any other type of code. The program also cannot handle files that exceed `Integer.MAX_VALUE` lines.

3.3 Problems Encountered

- Our initial solution design did not put individual commands in their own classes, which was problematic, since a lot of code was repeated. Thus, we changed it to our current OOP design.
- Infect was a special command, since it could take either 0 or 1 arguments, so we had to implement special functionality to handle both cases instead of just expecting only one case.
- We didn't realize that any line jump could be specified in absolute, relative, and register terms, and only thought that it was applicable to `go`. Instead, we had to go back and implement functionality for any jump command doing so, using special parsing instructions.
- We initially didn't have a way to store whether or not a jump was absolute, relative, or a register, so we had to implement flags to carry that information.

3.4 Limitations

- Our program does not provide support for critter code files that are longer than `Integer.MAX_VALUE` lines long.
- Our program only works when the Critter interface is implemented properly, otherwise it may break.
- Our program is not able to detect infinite loops or other undesired behavior within critter codes, and is only able to execute them.

3.5 Code Repeatability/Quality

Our program utilizes the concepts of OOP heavily and is able to reuse a large amount of code. All the parsing and processing for any critter command is relegated to a specific class, and all commands with the same types of arguments use the same function to process their input. Also, our code is easily extendable, as more commands can be added easily and our code should be able to immediately accomodate them, since the `executeCommand` function processes all commands through the abstract `Command` class, from which all commands extend.

In addition, we included a plethora of input validating measures to make sure our program behaves as intended and deals with unexpected input, including but not limited to:

- validating input file is not null
- validating each command line starts with a valid command
- validating that each command has the correct types and amount of arguments
- validating that integers are in the correct format and are parseable
  - There were many special cases in parsing an integer. For example, "r02" should not work, but parsing the integer part, "02" is perfectly valid. We must check for situations similar to this.
- validating that parameters are not null

3.6 Potential Improvements

- Implementing a critter code compiler to go along with the interpreter, so that critter programs are easier to write
- Implementing functionality to detect infinite loops or other undesired behavior before it happens during runtime
- Implementing functionality for longer length critter codes

4 Testing

4.1 Motivation & Observations

For testing, the classes we needed to test were split into a couple major categories: the interpreter itself, the static utility classes for parsing input, the abstract command superclasses, and each individual command class. We had to address two major issues: testing abstract classes and testing methods that need access to a critter (since we don't have access to a concrete critter implementation). We fixed these problems by importing an external package called Mockito, which allows us to "instantiate" abstract classes without needing to make them concrete. Then, we can call any concrete methods from the mocked class, and also manually specify return values when abstract methods are called, such as `getCellContent` in Critter. Plus, we can check if a certain Critter method is called. Combining this `verify` function from Mockito and asserts from JUnit, we can automatically check testcase correcteness. This allows us to test all of our classes and methods, controlling all inputs to the program and monitoring all outputs, so that we can exhaustively test each of our classes.

4.2 Black Box Testing

Black box testing wasn't as prevalent as white box testing since we had a lot of custom classes and methods that we needed to test individually. However, we used black box testing for our interpreter, because it dealt with the list of critter code and the entire input file. Thus, we opted to process/execute a variety of critter files, in combination with Mockito, that included all variations of commands. We checked to make sure that the correct Critter methods were called and that the correct line changes were recorded for a variety of commands, which should cover any case that can come up in a critter program.

4.3 White Box Testing

Most of our testing involved using JUnit to run automated unit tests on every single one of our classes. We used these tests to check the internal state of our many command classes, ensuring that both the loading and execution phases of each command operate as expected:

- Interpreter: we automated tests on `loadSpecies` and `executeCritter` as well as a helper function, checking to make sure correct commands were loaded in and execution stopped in the correct places; We used IntelliJ's debugger mode as well as print statements to make sure that commands were executed in the right order, and stored in the correct order/format inside our list of commands.
- utility classes: we automated tests with a variety of both valid and invalid lines of critter code, covering pretty much any case, including all the valid commands as well as commands with invalid syntax, arguments, or formatting

- command superclasses: we made sure the input was parsed correctly from a given string
- command classes: we made sure the correct methods were called to critter and the correct next line number was returned for all types of jumps, commands, and critter behavior

4.4 Edge Cases

- File has no lines of code: we make our list of commands empty
- Trying to jump to a line greater than `Integer.MAX_VALUE`: we assume this is out of bounds and do nothing for the rest of that critter's turns
- Jumping to a negative line number: out of bounds behavior, do nothing for rest of turns
- File has a blank name for the critter: the stored name will be blank
- Dealing with modifying stored register values near `Integer.MAX_VALUE` and `Integer.MIN_VALUE`: we default to Java's native overflow behavior, assuming the registers to just be regular integers.

5 Our Critter

5.1 Design

Our critter was designed to be essentially an improved rover, with the following key differences:

- If an enemy is at a 45 or 315 bearing, then turn in their direction - more chances to eat
- Only turn right when blocked by a wall or ally - removes randomness and thus critter doesn't stall (interestingly enough, either turning right or left all the way through didn't work as well as only turning right)
- Prioritize infecting enemies when our numbers are low, and prioritize eating when our numbers are high - influenced by a benchmark that changes based on how many enemies/allies this critter has seen (we found a high benchmark, with a high flexibility for change works the best)
- Prioritize infecting enemies early game, and prioritize eating late game - influenced by a benchmark that counts the number of moves the critter makes
  - Infecting full enemies during early game will give the infected critters more potential to do damage.
  - Eating hungry/starving enemies late game will give you food. Infecting is bad in this situation because then the infected critters just act as food for enemy species.

5.2 Results

Our Critter can consistently beat the Rover at a 1:10 ratio. The best result we've gotten is a 1 vs. 200 win, but it occurs quite rarely.

Since we didn't have time to write a critter compiler, one strategy we employed for making critter code easier to write was including a bunch of dummy lines in the file so that we could replace the dummy lines with code as necessary without shifting all the line numbers off. These lines could be removed after we finished the critter.

6 Social Impact

Critters competing in Critterfest is undoubtedly an entertaining experience, but just enjoying critters battling it out misses the most important detail - all the work that goes into building a good critter. In particular, the competition format serves another purpose - providing motivation for people to create the best critter possible. This theme is also ever present in the wider field of artificial intelligence, with some of the most powerful AIs being created as a result. For example, the well known AlphaZero is a direct product of trying to create an AI that beats out other contemporary chess AIs as well as the best human players, which resulted in a staggeringly strong AI that only lost 6 games out of 1000 to Stockfish, the standard for a modern chess engine (https://www.deepmind.com/blog/alphazero-shedding-new-light-on-chess-shogi-and-go). Plus, any industry grade AI, such as social media companies' facial recognition algorithms, are also born directly out of competition - not with other AIs directly, but with competition against other social media companies to produce the best product. Competition leading to better and better AIs is a distinctive feature of modern technology, and it doesn't seem to be changing any time soon.

7 Pair Programming

7.1 Log

Name	Time (hours)	Description
A & B	2	Solution Design
A	3	Driving
B	3	Driving
A	3	Individual
B	3	Individual
A	2	Driving
B	4	Individual
B	2	Driving
A & B	2	Testing Design
A	3	Driving
B	6	Individual
A	6	Individual
A & B	2	Critter Design
A	5	Driving

7.2 Experiences

7.2.1 Scheduling

Since my pair programmer and I are roommates, scheduling wasn't really an issue. It was actually quite convenient because we were able to work on our project together any time. We designed our class structure on the walk to dinner, and by the walk back, we had the entire structure planned out. It also helped that we were able to work on our project for long periods of time, being able to switch easily from individual development to pair programming.

7.2.2 Driving + Navigating

Most of the time, we were able to find low-level errors pretty easily. There were times when a couple of logical errors flew by our heads, but we eventually fixed them through testing. When we ran into a problem regarding our high-level design, oftentimes the driver would step back, and there would be two navigators. During pair programming, driving is quite nice and is much better than individual work because it's like having an extra brain attached to your own. You really don't have to think too much about what you're actually coding, whereas individually, I noticed I would have to take gaps to recount what I was trying to do.

7.3 Issues

One issue we had was managing merges and pulls over git. In the beginning, we encountered numerous merge conflicts, until we finally made a plan to allocate which classes were for who. Setting up the `.gitignore` file was also annoying because there were many files to ignore, and then some more to create exceptions to. Whilst we were working, we would sometimes encounter design differences, but they were usually resolved after some deeper understanding into their pros and cons.