**Design and Post-Mortem**

A. The top priority overall for the attacker is to gather all the power pills, preferably in the order of nearest distance. Let defenders that are not vulnerable be called aggressive. Any time after a power pill is consumed and there is at least one (1) vulnerable defender, the attacker will seek out and destroy the nearest vulnerable defender. There are specific precautions that are taken first. If there is an aggressive defender in between the attacker and the nearest vulnerable defender, the attack will ditch and run in the opposite direction. If there is no hostile obstacle, then the attacker has the new top priority of chasing down the nearest vulnerable defender. This continues until all defenders are aggressive, and the top priority switches back to eating power pills. Under the condition that the attacker encounters the nearest aggressive defender within a set gap of 4 units, the attacker goes for the nearest safe pill. There is a safe pill list that is a sub-list of the pill list only containing pills that poses a safe direction for the attacker to turn to. Lastly, if being actively chased by an aggressive defender within the short preset distance gap, the top priority shifts from finding power pills to simply running away to safety.

A close up of text on a white background

Description automatically generated

B. The first major and noticeable problem I encountered from the beginning was the seizures that gator was suffering from. After many hours, I discovered several factors. Firstly, if either 2 or more power pills, regular pills, or closest aggressive defenders were equidistant to the attacker, the code has trouble picking the nearest one. This causes indecisiveness to the direction to turn next. This was fixed by sorting the pill and power pill lists by the distance to the attacker, and then the first element was picked (using get(0) method from List) as the closest element. Also, the attacker was wasting time not chasing any vulnerable defender after eating only one (1) vulnerable defender. The problem lied in an erroneous if statement that makes the attacker pursue only the nearest defender after eating a pill and nothing more. This led to the attack not eating other vulnerable defenders and sometimes die by mistaking aggressive ones as vulnerable. This was fixed by divvying up the if statement into an if-else-if chain. This time, attacker chooses whether to pursue the next closest vulnerable defender depending on whether the nearest defender overall is aggressive or not and its relative location. Those are the major issues encountered. From the start, there were several things that were already beneficial for the attacker. A copy of the pill list was shortened, via stream and filtering operations, to a smaller list for safe pills. The next direction from the attacker to each safe pill does not collide with going towards an aggressive defender. This protective measure allows the attacker to flee the nearest aggressive defender while simultaneously having a chance to gain points along the way. The best part about the implementation of the update method of the student controller is the prioritization order of actions based on current conditions. The attacker always attempts to eat a vulnerable defender, therefore using vulnerable time wisely. Otherwise, try to eat more power pills and repeat. As a last countermeasure, avoid the nearest defender and eat safe pills.

C. The project became a great practice for logical reasoning and algorithm design in a way. I was able to focus on finding an optimal order of actions and priorities to gain as much points as possible for the attacker. This was also a solid time for practice with debugging. The most important part of the project was increasing my expertise in highly useful standard Java functions, especially using stream to sort, filter, and manipulate lists of objects. For example, the safe pill list is formed by filtering out any regular pill such that the next direction from the attacker to said pill leads to an aggressive defender. Since I had quite some code for a single function, I also seized the chance to use Java 8 parallelization on streams to reduce the strain on the processors. This allowed each tick in the game to be updated seamlessly despite a lot of processing needed.