# The Basics

In our Game Of Life simulator, we decided that we wanted to offer a variety of concrete creatures (Antelope, Bear, Bunny, Yeti, Trex, and Honey Badger). The user can control how many of these are created via a GUI, and each creature would be able to move just 1 space, with some preferring to go after other creatures, and some prefer to flee. Each turn, the animals have a set of steps they always take. They look around, try and find nearby animals. They then move based on movement strategy. After moving, the animal may fight another creature if it lands on a tile with them. The animal then tries to eat if the tile has food (more on tiles in a bit.) If there is food, the animal regains some energy. If there is no food, the animal loses energy based on a BurnCalories method. **Ideally this would be based on a metric of Weight of Animal verse actions taken and food consumed.**  But in this case, we had to cut some corners.

Tiles are created when the game board is generated. These tiles have various properties, and most importantly food. This determines if an animal can eat.

We have the structure to create natural disasters that can destroy Animals, and damage tiles (which could regenerate over time) but this feature was also something that was cut out due to time.

# Patterns

We implemented a singleton for the Animal Factory, to ensure that there was only one Animal Factory. This AnimalFactory implemented parts of the Abstract Factory Pattern, and the Composite builder Pattern. The Factory creates different concrete implementations of Animal based on We used the composite pattern to implement individual body parts that make up a whole animal. These body parts all have special attributes in their own right which is why we used the composite pattern in this fashion. These individual parts include a head, arms, legs, and torso, with a body class. We have implemented an iterator pattern that adds the weight of all the animals together. We also implemented a weight decorator pattern to make different body parts weigh a different mass based on the decorator value applied. This decorator pattern ensures that the individual object only is changed and does not affect the behavior of other objects from the same class.

We have five different types of terrain in the form of tiles throughout the grid. These come in Desert, Tundra, Mountain, Forest, and Water. Each tile has certain properties attributed to it such as the probability of finding food to restore energy. For the tile creation, we implemented a builder pattern. We thought the builder pattern would be a good choice here considering that our tiles have many parameters to take in. Our tiles are randomized throughout the board. For our abstract factory pattern we invoked the idea of a natural disaster factory pattern to try and be a little creative. There is an array of different patterns which do different things to the game board. For example, one pattern can make a non-desert tile turn into desert by wiping out all the substance in it. Another disaster could add a certain amount of toxicity to a tile that will harm an animal if it approaches too closely. We also implemented a flight or fight strategy pattern where animals who have the flight attribute will move to a tile which is unoccupied by an animal and vice versa for fight. If two animals engage, the one with the most strength wins. Each time an animal moves and does not find food a certain amount of health is lost each turn. The bigger the mass of the animal, the more energy lost per turn with each animal starting at a base health of one hundred.

# Execution

To run from Netbeans, Run the **SettingsGUI.java** file from the cs407final.GUI package.

When you run the program it will first come up with a GUI in which you can fill in different properties of the game such as the tile size, chance of natural disasters happening, and how many of each animal you want to produce. When you are satisfied, you can then hit the play button to watch the simulation run. Messages will print out to the console showing what each animal is doing on the current turn. You can play out as many turns as you want within the simulation provided there is at least one animal alive.

# Other Notes

The User Created Animal functionality is nearly 90% complete. There was some issues parsing the files coming in, and looking back it would have been better to do this via JSON files, than comma delaminated files. Not sure where the trouble is with this, but somewhere along the lines it blows up. The idea was to accept the file in, read it line by line, and pass these into the Overloaded method createAnimals() , which will create a custom animal.

Would really have loved to have done more with the Natural Disasters, but they are currently not affecting anything in this.