Title: Saskatchewan Animal Sim		
Authors: Paul Hewitt, Taylor Petrychyn, Maksym Zabutnyy	Date: 31/10/16	ENSE 374

Revision History:

Rev:	ECO#:	Change Description:	Approval:	Date:
P1	-	UML & Empty Classes		01/11/16
P2	-	Saskatchewan, Animal Generation		9/11/16
P3	-	Display and Main Menu		9/11/16
P4	-	Vegetation Classes, Plant Generation		13/11/16
P5	-	World Generation and World Tick		26/11/16
P6	-	Movement, Collision, Hunger		27/11/16
P7	-	Properties of Animals/Plants		27/11/16

Table of Contents:

1. Introduction
2. Revision 1
3. Revision 2
4. Revision 3
5. Revision 4
6. Revision 5

Introduction:

This project mirrors the wildlife food chain of Saskatchewan using a simulation built using Java. We were presented with a diagram that connects the various animals and plants that live in Saskatchewan and the group was to create the transferable relationships to code. Our robust program includes various hierarchal classes that the animals and plants inherit from, as well as, classes for display, entity generation, and our main, saskatchewan. The goal of our design is to create a grid that simulates the behaviour of the animals in a habitat using various attributes and show their progression over time based on the constraints of the food chain.

Revision 1: UML & Empty Classes

To begin our process, we created a UML diagram that transferred the concepts of the food chain to a software design principle. Our first iteration of the UML diagram showed STUB classes that were easily coded in Java. We started by creating our two large ‘parent’ classes, Animal and Vegetation. All living things in our simulation would inherit from one of these two classes. Next we created three additional classes to describe animal feeding behavior. These were Carnivore, Herbivore, and Omnivore. These three classes inherited from Animal (Herbivore extends Animal). As our vegetation did not have any feeding habits, we did not need to include any additional parent classes.

Finally, we created a class for each individual animal and type of vegetation, and those classes inherited from their respectable feeding habit classes. Fox inherits from Carnivore, and Grasshopper inherits from herbivore, etc. This seemed like a lot of work for the initial stub programming, but we created a flexible interface that allowed us to control almost all aspects of animal behavior utilizing inheritance.

Revision 2: Saskatchewan, Animal Generation

Our next step was to create our class where all the action would take place. We appropriately named this class Saskatchewan. For the most part this was left blank, but it would be the class that would contain the world display, world tick, create the entity arrays, etc. Essentially the Main of our simulator. Next we tackled basic entity generation, specifically animals. We decided to just focus on animals, as adding vegetation would be easy once all the groundwork was completed. We made a class called Animal Factory, and it contained a basic switch statement with each case representing a different animal. This would allow us to select a new random animal by calling a GetRandomAnimal function (picking a random case) contained in our Animals class. Once we had the ability to pick animals at random, it was time to create our display, and place animals inside of it.

Revision 3: Display and Main Menu

For our ‘map’ of Saskatchewan, we created an array with constant rows/columns, and filled it with ‘x’ to indicate an empty space. For our testing, the array is only 20x20, but it can be scaled very easily. We chose to make it smaller, so there would be a higher chance of different animals encountering one another. To represent the different animals, we used the corresponding first letters of each animal name. We had to come up with a way to differentiate between animals and vegetation, (as Grasshopper and Grass would both be G), so we gave animals capital letters, and planned to give vegetation lower case letters. When we randomly placed an animal within the map, the ‘x’ occupying the desired coordinate would be replaced by the capital letter.

We began to implement a main menu to compliment our display. That is, a way to control the simulation. We wanted to have three main commands, add an animal, add a plant, and simulate. By simulating, it would give all the entities a chance to move around, and eat. We planned to add features to make managing the simulation a little easier too. Things like auto populate so you don’t have to manually spawn entities, or a command that would simulate for ten ‘turns’ and then output the results. We didn’t want to force the user to type one hundred commands before they could even run the simulation. These features were not implemented immediately, but planned for future revisions.

Revision 4: Vegetation Classes, Plant Generation

This was by far our easiest revision, as majority of the groundwork was already completed. We simply had to replicate our animal spawning, but use vegetation instead. This was mainly comprised of copy and pasting code that we had already written, and slightly modifying it to work with our existing vegetation classes. As of revision 4, we had completed all required tasks for milestone one.

Revision 5: Movement, Collision, Hunger

In this revision, we found out if we planned and designed our project correctly, as this is where all the functionality was added. Overall, we found that our design was just about where we wanted it to be, only having to add roughly three functions to the Animal class in order to flesh out movement, collision, and hunger. From a design perspective, we were quite pleased, as we didn’t want to have to redesign the majority of our classes. For animal movement, we generate a random number depending on the animal, and add or subtract it to the corresponding x and y coordinates. Certain animals may move faster than others; rabbits and birds can move faster than the fox.

When animals collide, they check which object the animal is from. Each animal class has a CanEat member function, that lists all the things they are able to eat. Furthermore, each animal has a hunger and feed value. When hunger reaches 0, that animal starves to death. If an animal eats another, the prey’s feed value is added to the predator's hunger value, thus keeping it alive.