Project 1 Design Proposal

Document Author(s): Scott Spencer

Date: Fall 2016

Design Rationale

Describe what design decisions you made and why. Convince your manager that your design can lead to a successful implementation of the requirements.

For this ecosystem program I have decided to use a total of seven classes (including the provided GUI) with one interface to implement the behaviors required by the assignment. These classes are called: *EcoEngine*, *EcoIO*, *Animal*, *HighAnimal*, *MidAnimal*, *LowAnimal*, and *EcoGUI*. The interface is called *AnimalIsPrey.* The diagram below shows these classes/interfaces and their interactions in a UML class diagram. The *EcoGUI* class is responsible for taking input from the user and providing formatted output. This class has an instance of *EcoEngine* class and calls *EcoEngine* methods to read what is defined in the grid as empty and full spaces, steps of the program, and how the objects of the program are meant to interact with one another. To help in doing this, *EcoIO* reads in two .txt files, one that populates a grid of animals and sets the type, species, and position of these animals in the grid and another .txt that provides configuration information for these animals if the initial default grid is not the desired output. *EcoEngine* has an instance of the **abstract** parent class *Animal* which is used as a data type for the different types of animals which are split into the child classes. *Animal* also implements the interface *AnimalIsPrey,* which ensures the child *classes HighAnimal, MidAnimal, and LowAnimal all have the common methods meant to provide an Animal's* behaviors, which are outlined in that interface. *HighAnimal* is the child class that extends *Animal* and defines the behaviors specific to animals that are considered high on the food chain. *MidAnimal* is the child class that extends *Animal* and defines the behaviors specific to animals that are considered in the middle of the food chain. *LowAnimal* is the child class that extends *Animal* and defines the behaviors specific to animals that are considered low on the food chain.

<Diagram included in "UML >

The *EcoGUI* starts the program and uses calls to the class *EcoEngine* to retrieve the behaviors of the simulation. *EcoEngine* will call an instance of *EcoIO* in its constructor method in order to read in two files whose filenames are passed as parameters from the GUI. The first file is to build the default ecosystem grid with default animals and their positions, and the second is to provide the configuration for those animals (their rank, color, starve step number requirement, and breed step number requirement). From these, two arrays are built, one with strings of the species of animals found in the grid from the first file, and the second an array of characters from the same file that holds the symbols for those animals. From here, *EcoEngine* runs its constructor with a parameter of the steps to be run (passed from the GUI), which is used to determine how many times *EcoEngine* runs a loop of its method calls. Within *EcoEngine's* loop is an integer called "i" that will be used to pass the current step number to other method calls as parameters that will aid in determining the age of animals in the system. *EcoEngine* calls to *EcoIO* in order to help it construct a 2D array of animals that will act as our representation of the current grid of animals in their indexed positions. *Animal* is an **abstract** parent class that will implement the **interface** *AnimalMoveBreed. Animal* has 3 sub or "child" classes by the names of *HighAnimal, MidAnimal,* and *LowAnimal* all of whom have fields and characteristics that set them apart from the others, but who the methods outlined in *AnimalMoveBreed* and *Animal.* The *AnimalMoveBreed* interface ensures all instances of *Animal* check whether the cell in question is empty, and checks whether a non-empty cell is prey. *Animal* is instantiated with parameters in this array through *EcoEngine* and contains boolean fields for whether this animal instance has **eaten, bred, moved,** and is **disabled.** It also has fields which keep track of the *Animal's* **xPosition, yPosition, species,** and **birthStep** (which I will use with the current step to calculate age). The *Animal* class will make use of several boolean methods in conjunction with conditional statements to determine the outcome of the "animalActs()" void method that takes in the 2D array as a parameter and points to a cell in the grid (which then moves clockwise until the appropriate action/cell is found). These methods return if the space in question is empty, whether the animal will breed, whether the animal (TO BE EATEN, not "this" animal) is prey, whether the animal moves to the cell in question, and whether the the animal dies. *HighAnimal* has fields "hS" and "hB" which are integers that determine the number of steps before the

animal will starve (“hS”) and how many steps the animal is required to take before it can breed or breed again (“hB”), it also includes a method for eating since that particular piece of their actions differ from the other types of animals. *MidAnimal* has fields “mS” and “mB” which are integers that determine the number of steps before the animal will starve (“mS”) and how many steps the animal is required to take before it can breed or breed again (“mB”), it also includes a method for eating since that particular piece of their actions differ from the other types of animals. *LowAnimal* is slightly different in that it cannot eat other animals on the grid, and dies of old age instead of starvation. *LowAnimal* has fields “lS” and “lB” that are used to determine how many steps before the animal dies of old age (“lS”) and how many steps are required before the animal can breed/breed again (*“lB”).*

Figure 1: Class Diagram for <Project Name>

Document Revision History

Date	Author	Change Description
09/21/2016	Scott Spencer	• Created file, wrote abstract