COMP 9 Lab 6: Agent-Based Simulation

Due: Thursday, March 31st, 12:00 PM

March 16, 2011

1 Introduction

Your Tamagotchi game was a *simulation* of a pet, but your pet only interacted with the player of the game, via `gets` and `puts`. In this lab, you will write a more sophisticated agent-based simulation, in which multiple instances of your pet class will interact. The *agent* in this simulation will be whatever sort of pet you developed for Lab 5. In essence, you're taking your pet and turning it into a lab animal!

2 Requirements

- Develop several methods by which your pets can interact with each other. These methods should satisfy the basic needs of your pets: food (do they feed each other, or eat each other?), play, sleep, and so on. Add these methods to your class definition (`pet.rb`)
- You may use the pet class (whether you called it `Pet`, `Dog`, `Koala`, or anything else) you developed for Lab 5. If you'd rather develop a new pet class, you are certainly free to do so.
  - None of your methods should `exit` the program.
- Come up with some new methods that involve *creation* or *destruction* of your animals. Examples might be:
  - breeding
  - hatching
  - being eaten
  - dying of old age

- Develop a *container* class to hold your pets. Your container could simply be an array, but wouldn't it be convenient to be able to say something like `farm.animals.number_alive` or `pigpen.pigs.feed`? Therefore, you must write a container class, though it can *use an instance variable* that is itself an `array` to store the animals.
  - Your container class should live in its own file, `container.rb`, which should be `required` by your `pet.rb` file.
- Have some way to track which animals are alive.
  - an `@alive` instance variable?
  - simply remove the animal from the *container*
- Write a simulation program, `simulation.rb`, which creates one container class and populates it with some initial number of animals
  - Your simulation should ask the user how many *time steps* he or she would like it to run for (using `gets` and `puts`)
  - Your simulation should run through a number of *time steps*
  - after those time steps, it should report how many animals are alive, and some information about them (such as age or health; the specifics of this are up to you). At this point, the program can exit.

3 Hints

3.1 Removing objects from the collection

In Ruby, given some collection (say, an array) of objects, all we have to do to make the object go away is to remove it from the collection:

```
spot = Dog.new('Spot')
fido = Dog.new('Scooby')
spike = Dog.new('Lassie')
collection = [spot, fido, spike]
collection.delete(spot)
# now collection == [fido, spike]
```

This could be done from *inside* a class, but the class has to know about the collection:

```
class Termite
    def initialize(colony)
        colony.add(self)
```

```
        @colony = colony
    end

    def die
        @colony.remove(self)
    end
end

class Colony
    def initialize
        @collection = []
    end

    def remove(insect)
        @collection.delete(insect)
    end
end
```

3.1.1 What would the `add` method in the Colony class need to do?

(answer this yourself)

3.1.2 What happens to the termite after it dies? It's removed from the colony, but what happens to it?

Answer: it's *garbage collected*. Objects that cannot be reached from anywhere else in the program, because no other variables refer to them, are automatically deleted.

3.2 Interacting with another object

How could one `Dolphin` talk to another?

```
class Dolphin
    attr_accessor :squeaks_received
    # initialize method hidden for brevity
    def squeak(other)
        @squeaks_emitted += 1
        other.squeaks_received += 1
        other.squeak(self)
    end
end
```

So, we have a class of `Dolphin` that perpetually squeak at one another, and count the number of squeaks they've sent and received. Not terribly useful, but this should illustrate how objects can communicate with one another.

3.3 Handling time steps

You should still have a `passage_of_time` method to indicate how your *instance variables* change from one time step to the next for your animal. However, to keep your animals in sync, you should have a `tick` method for your *container class* that calls `passage_of_time` on each animal in the collection. This way, they all age, get hungrier, and so on, even if none has been interacted with during that time step.

4 Design Document

Please answer the following questions:

1. What will your container class be called?
2. What new actions will you add to your pet class to allow them to interact?
3. How will your container class handle adding and removing animals?
4. What will you measure after the user-specified number of time steps?
5. What methods will you need to add to your *container* and *animal* classes to accomplish measurement?

5 Handing in your solution

When you are satisfied with your program, submit it to be graded. Note that all three files will be needed:

```
provide comp9 lab6 pet.rb container.rb simulation.rb
```

Make sure to hand in your design sketch when you're done with it.