Singleton Pattern

CS342 Fall 16

Classwork

Datalog

- TBD

Solutions?

- Create a DataLog global object and hope it doesn't get overwritten
  - But what if you accidentally create two data log objects?
- Create a datalog object and pass it to every method
  - Seems tedious and error prone
- Create only one object and guarantee that no other object will be created
  - how?

Why not Globals?

- You can define a global variable in Ruby with $
  - $global_var
    - The variable is now available throughout your program
- Why not use this instead?
  - No lazy instantiation
  - No protection
    - You could accidentally set $global_var to 1

The Singleton

- A Singleton is a class that can have only one instance and that provides global access to that one instance.
  - Without using global variables
- The singleton object manages the creation and access to its sole instance.
  - The first of our next series of Creational Patterns

Class Variables in Ruby

- Recall class variables are attached to a class, not an object
  - static variables in Java
- Create a class variable in Ruby by adding another @ symbol in front of the name
  - The following class keeps track of how many objects have been created
    - class ClassVariableTester

```
@@obj_count = 0

def initialize

    @@obj_count++

end
```

Class Methods in Ruby

- Class methods are a little trickier
  - When you are outside of a method, in your class, self refers to the class, not the object
    - self becomes a static variable
      - the shift is dynamic based on context
  - For example:
    - class SomeClass

      puts("Inside a class def, self is #{self}") #will print out the class name

      def print

      puts("Inside a class def, self is #{self}") #will print out the object id

Defining Class Methods

- define class methods with the ‘self’ keyword
  - def self.class_level_method

    puts('hello from the class method')

    end
  - You can also just call out the class name
    - def SomeClass.class_level_method

      puts('hello from the class method')

      end
- Call a class method using the class name
  - SomeClass.class_level_method

Creating Static Instances

- We can make the class responsible for creating and managing the instance object
  - Add a class variable to the datalogger
    - @@instance
  - Create a class method to return the instance
    - def self.instance

      return @@instance

      end

Using the static instance

- Whenever we want to get the logging instance, we just call the class method
  - logger = DataLog.instance
  - logger.info(“Hello World”)
- Better yet, just call the instance when needed
  - DataLog.instance.info(“Hello World”)

Eliminating rogue instances

- A requirement of the singleton is to ensure that the one and only singleton is the sole instance
  - What is someone does this: DataLog.new
- Make the new method private to keep objects from being instantiated
  - private_class_method :new
    - This works because new is a class method

Ruby makes the singleton easy

- The built-in Singleton module turns any class into a singleton
  - require ‘singleton’
  - include Singleton

Eager vs Lazy instantiation

- Eager Instantiation Singletons create the single instance immediately
  - Our DataLog is an eager instantiation singleton
  - @@instance = DataLog.new
- Lazy Instantiation Singletons wait until the class instance method is called before creating the object
  - The ruby ‘Singleton’ module uses lazy instantiation
    - def self.instance

      @@instance || @@instance = DataLog.new

      return @@instance

Security in Ruby

- Ruby is such a dynamic and open language, how do we ensure security?
  - How can we prevent a user who wants to create another Singleton object from doing so with Class Runtime Modification or calling clone
    - We can't
- Ruby's philosphy:
  - The language tries to prevent you from making accidental mistakes, but won't get in your way if you are determined to make them
  - This has serious security implications

Where is security?

- Security truly rests on your program's inputs and outputs
  - primarily inputs
- It is your job as the programmer, to use good input practices to avoid security risks
  - The language shouldn't have to think for you
- Ruby's security model is that the programmer should write secure code, not the language

Problems with the Singleton

- Singleton is probably the most hated, derided of all the patterns.
  - Why?
- Leads to tightly coupled classes
  - Tightly coupled classes are dependent on one another's state
- Makes unit testing difficult
  - Unit testing depends on state, and singletons have ever changing state throughout the life of the program

Classwork

Hot Beverages

TBD

Classwork:

Poker Night

- TBD