Name: Jamie Penner Date: 11/25/19

Lab 7 – Singleton Pattern

Objectives

Part 1 – Setting up the SingletonLogger

Part 2 – Developing the SingletonLogger

Reflection

Background/Scenario

As we have discussed in class, it is sometimes necessary to have a single shared resource for an entire application. In this lab, you will be implementing a thread-safe file logger that exercises a double-checked locking Singleton pattern as one example of that.

The SingletonLabStarterCode.zip demonstrates how to create five separate threads that start execution at the ExerciseThreads.ExerciseLogger method. Review the code for familiarity. You will need it throughout the lab.

Required Resources

- Visual Studio 2017

Part 1: Setting up the SingletonLogger

Step 1: Download the SingletonLabStarterCode solution

Download the lab starter code solution from Canvas and add it to your repository. Look around the code and familiarize yourself with the project.

Step 2: Create a new Class Library

Add a class library (dll) to the starter code solution and call it SingletonDll. Add the namespace and a reference to your dll to the existing ThreadingDemo project.

Step 3: Create the SingletonLogger class

In your dll, rename class1.cs to "SingletonLogger" and implement a thread-safe (with double checked locking) Singleton pattern for controlling the object lifetime and access to the single instance.

Provide a basic public LogMsg() stub that can be used for centralized logging. All logging should be directed to a file named "log.txt." Since log.txt will be unchanging static data, it is acceptable to make this a constant field in the class. This would be broken out into the application's configuration, but that is an optional requirement here.

In order to ensure that messages are not interleaved, the LogMsg() method needs to serialize each write to the file. Add Thread.Sleep(10) after the actual logging operation and before releasing the lock to simulate a large volume of actual data logging.

Part 2: Developing the SingletonLogger class

Step 1: Instantiate a private StreamWriter

Declare a private data member “streamwriter” of type *StreamWriter*. (You might need to add the System.IO namespace) Pass in the name of the file you are going write to as a string parameter of your private SingletonLogger constructor. Initialize the *StreamWriter* data member in your SingletonLogger constructor.

```
streamwriter = new StreamWriter(File.Create(LogFile));
```

Where “LogFile” is the string passed into *the SingletonLogger* ***constructor***. (Even though the constructor is private and log.txt is hard-coded, we can still design the constructor to follow OCP and DIP by parameterizing.)

Step 2: Create the LogMsg method

Implement a public “LogMsg” method in the SingletonLogger that takes a string as a parameter and outputs it to the Logfile along with the current date and time, similar to:

```
streamwriter.WriteLine("{0} - {1}", DateTime.Now, msg);
Thread.Sleep(10);
```

Now we need to ensure that the writes to the LogFile are serialized because the file I/O methods are not threadsafe. Not protecting them with some kind of lock will result in an exception and best and data corruption at worst.

Add a new private data member at the top of the SingletonLogger class and use it to serialize access the LogMsg() code. Treat writes to the disk as a “critical section” that only one thread can access a time. That thread requires ownership of the lock to enter; other threads that come along and want access will have to wait until the lock is free. Use the lock(<object>) scope to define the boundaries of the section of code that must be thread-safe. (Note: Use a different variable for the lock object than the one used in SingletonLogger.GetInstance().)

```
private object _streamsync = new object();
```

Step 3: Implement the Close method

Implement a public “Close” method in SingletonLogger that closes the file. Call it from programs.cs after all the threads have joined.

Step 4: Update the ThreadLogger method.

In the program.cs file’s ExcerciseThreads.ThreadLogger() method, change the Console.WriteLine statement to a statement that calls your SingletonLogger’s LogMsg method passing it the following string:

```
String.Format("Writing Message {0}", nIter);
```

Step 5: Take performance metrics on the application.

Before starting the threads, create and start a timer (Stopwatch). (You will need to add namespace System.Diagnostics to program.cs.)

After the threads have all completed, and the SingletonLogger file data member has been closed, output the elapsed time to the console.

Step 6: Run the application.

Run the application. All should work. Check out the log.txt file to see if all messages are logged. Note: the log.txt file will be in the ThreadingDemo->bin->debug directory.

Reflection

1. Why use two different locks for GetInstance() and LogMsg()?

   So there aren't any thread collisions when going through instance or LogMsg. If there was only one lock for both, the program may queue one of the threads thinking a thread is already inside of the lock for that function when the thread is actually inside of the other lock for the other function.

2. Why is the lock object variable in GetInstance() static and the lock object variable in the LogMsg() not static?

   We want the object for instance to retain it's value throughout the execution of the program. We should only have one instance of the SingletonLogger throughout the program. The LogMsg method has multiple instances throughout the program.

3. Why only call the SingleLogger.Close() method once?

   The file doesn't need to be closed until all of the messages have been written to the file.

4. Remove the lock in SingletonLogger.LogMsg(). Run the application. What happens and why?

   LogMsg is no longer threadsafe and multiple threads are trying to use the LogMsg method at the same time.

5. What is an Anti-Pattern?

   An anti-pattern is a commonly occurring solution to a problem that generates negative consequences.

6. Why is the Singleton pattern often considered as an Anti-Pattern?

   The use of Singletons makes it difficult to implement unit testing. Singletons also decrease performance, are difficult to extend and can create garbage collection and memory management issues.