Introducing a day in the office at the Atomic Pizza Machine

Act One

Developer 1: That new console logging system appears not to be 100% correct, sometimes log messages aren't in the correct colour. After close examination of the code it appears that it's a threading issue, were multiple threads are using the console at the same time.

Developer 2: So we need to ensure that only thread has access to the console at any one time. At the moment everyone creates an instance of the ConsoleLogger class, so adding monitor based synchronization to the ConsoleLogger won't work, this may need to change.

Developer 1: How about we create a static Global class that has a static field that points to the Console Logger. That way we can share a common instance throughout the application, thus monitor based synchronization logic can then be used inside the ConsoleLogger.

Developer 2: Hmmm that just smells bad...aren't global variables evil...and you aren't stopping someone from creating multiple instances of Console Logger just hoping they won't and use the global version.

Developer 1: You're right...Hey I've read a book recently on design patterns and there is this pattern called Singleton...It provides a means to guarantee there can only be one...I've called it the Highlander Pattern...

Developer 2: I'm not sure about this I've heard some evil things about Singletons.

Developer 1: How can code be evil?

Heavy refactoring to the singleton pattern.....

Developer 2: Ok so singleton pattern is similar to global variables, but with a tighter contract in that it guarantees everyone can only use a single instance.

Act Two

Developer 2: That's all working now....You know what I'm thinking that since we are in the mood for adopting new ideas and our code obviously isn't as robust as we first thought perhaps we should take a look at unit tests

Developer 1: Sounds like a good idea, let's give it a whirl...How about we try and write a test to ensure that when the Toppings Dispenser is asked to dispense we get the correct number of messages coming out onto the console...Doesn't sound too hard.

Create a new unit test project, and start crafting the test, but it's not too long before large amounts of hard scratching.

Developer 2: Seems we have a problem since in order to test that the dispenser is logging messages, we need to not use a true console logger, but a mocked one that simply records the number of times it was called, so that we can verify that after the dispensing method is called that the correct number of messages are sent to the console. Currently using your Singleton pattern means that we have no way to vary the implementation of the logger, inside the dispensing code. I told you this pattern was evil...but you wouldn't believe me...

Developer 1: Oh I remember one other thing from the patterns book it said you should design to interface as much as possible. This allows implementation to vary...So we need to write our code in terms of say IConsoleLogger .

Developer 2: All well and good but at some point metal has to hit the road and we need to somehow create an instance. I guess we could create an instance in our application init code, and pass around the reference to all the types that need an IConsoleLogger. I believe this is known as Inversion of Control.

Developer 1: Sounds cool...that may work for one or two objects but what if we start to need a few more objects like this, that is really going to be painful, especially if you end up having to pass objects through multiple layers...I think I can solve this with the Singleton pattern...

Developer 2: *Groan*....boy you like flogging a dead horse.

Developer 1: Run with me....rather than having the ConsoleLogger as a singleton, which clearly isn't good for unit testing as we can't vary the implementation, how about providing a means for code to discover at runtime what instance of ConsoleLogger to use. Were the instance is set by application or test initialisation, the object that provides this service I will call a Registry, and it will not only support ConsoleLogger but any other service the application requires. We don't want to be passing this registry around so I suggest we make it a singleton.

> *Creation of a Registry class and more frantic refactoring to use the registry, and develop the unit tests....*

Developer 2: Ok I can see how that works, but now we haven't stopped anyone from just creating another instance of the logger, and you know we do have some cavalier programmer s here that would just do that.

Developer 1: You have a fair point, so stepping back the implementation of the Registry class prevents multiple objects registering that implement a given service. So if we made our ConsoleLogger auto register on creation, via its ctor then any one creating multiple would get an exception. Since this functionality could be common, I'll place it into an abstract base class

> *Creation of a new RegistrySingleton<T> abstract class, and a bit of refactoring to utilise the notion of ConsoleLogger auto registers on first use of new...*

Developer 1: Phew...that works. See I told you Singleton wasn't that bad...look what we have managed to achieve.

Developer 2: Hmmm...

Act Three

Developer 2: I thought I'd better read up on the Singleton pattern a bit more since you are so keen on using it, and there is plenty of stuff out there on the World Wide Web that states its evil. So I have a challenge for you...Yesterday we wrote a single test that registered a mock implementation of the ConsoleLogger, let's write another test and this time use a different mocked Console Logger.

Developer 1: Sure, how hard can that be? What do you want this mocked logger to do?

Developer 2: Well it's not important for this test I just want a Null implementation of it, and just call a method on the dispenser.

Writing unit test with a Null ConsoleLogger...

Developer 1: Hmm looks like I've got a problem; I can only have one instance of the mocked object. When I run each test in isolation all works fine, but not together.

Developer 2: See you wouldn't listen Singleton is evil because its state is long running, and for unit testing we need to be able to run each test in a per test controlled environment.

Developer 1: Hmm you may have something...no wait how about I allow you to reset the registry instance.

Developer 2: You don't give up do you...But if you do that then you have lost the concept that the Registry is a singleton, since multiple instances could be out there.

Developer 1: Not if I make the Reset method private, and thus accessing it via code is not likely but it can still be accessed via reflection. Since unit testing isn't about performance this seems like a possible solution.

Adding a private Reset method to the Registry class, and invoking it via the unit tests.

Developer 1: Phew...that works...

Act Four

Developer 2: You know the other thing about singleton that I've read is evil is the fact that it doesn't follow "Single Responsibility" guidelines since the object is responsible for ensuring single instance and for its given piece of functionality.

Developer 1: Why such a big deal?

Developer 2: Well at the moment we can't see any reason why there would need to be more than one instance of the ConsoleLogger, but will that be the case in the future...Perhaps we will have a Console per machine in some of our more sophisticated shops. So we will need to allow instance control and implementation to perhaps vary.

Developer 1: Hmmm, ok well perhaps rather than registering a single instance with the registry why not register an object that is responsible for creating the object we require. In effect utilise the factory pattern, thus separating creation logic from object logic allowing both to vary.

Refactoring creating a ILoggerFactory and LoggerFactory class. Making ConsoleLogger an inner class of the LoggerFactory thus ensuring singleton creation.

Developer 2: Very cool...you know what we could use this mechanism not just for singletons but for anything. That way we could write all our application code in terms of interfaces and for all object creation delegate to a factory, were the actually factory itself could vary. This would be great for unit testing, allowing us to provide mock factories that create mock implementations.

Developer 1: Ok let's do that...

Refactoring The PizzaOven class to now use a IPizzaOvenFactory

Developer 2: You know I wonder if we need the Registry singleton at all now...why not just pass the Registry around, or better still a IRegistry, its after all only one object now..

Developer 1: I guess we could, but it would probably mean touching every object in our system to take a Registry, and I'm guessing we aren't going to move the entire code base over night to using this. I'm also concerned that whilst this is ultra flexible we will have to write a lot of abstract factories. Most of which will be pretty straightforward. I wonder if it could be automated.

Act Five

Developer 2: I've been reading around this topic a bit more, and it appears there may well be something that can automate this process for us called Dependency Injection Containers, they are very similar to your registry but they take care of the supplying the factories and controlling the lifetime management, saving us writing that tedious factory code...

Developer 1: Sounds very cool, you want to show me how it works.

Refactoring to use Unity IoC container.

Developer 1: Awesome, my only one concern is that it looks like **ALL** our code now needs to be aware of IUnityContainer, starting with a clean code base not a problem, but we have an existing codebase.

Developer 2: Singleton anyone?