# [Unity and the Hierarchical Lifetime Manager](http://chrismabry.wordpress.com/2012/05/19/unity-and-the-hierarchical-lifetime-manager/)

## Introduction

When registering types with the IUnityContainer, you may specify a LifetimeManager whose purpose is to scope the lifetime of any instances resolved out of your container for that type. The purpose of this article is to discuss the utility of one of these: the HierarchicalLifetimeManager.

## How Lifetime Managers Work

When using the RegisterType method on the Unity Container, the default lifetime manager is the TransientLifetimeManager. This lifetime manager returns a new instance of the registered type on each request, which is not the behavior I want here. Instead, I am going to explicitly specify the ContainerControlledLifetimeManager. By using this lifetime manager, any resolution of the registered type will give me a singleton instance which is scoped to the lifetime of my container:

IUnityContainer unityContainer = new UnityContainer();

unityContainer.RegisterType<IDiagnosticsManager>(new ContainerControlledLifetimeManager());

IDiagnosticsManager diagManager = unityContainer.Resolve<IDiagnosticsManager>();

There is a third lifetime manager I want to discuss, the HiearchicalLifetimeManager. This LifetimeManager resolves a singleton instance of the registered type scoped to the lifetime of the container which performed the resolution (but not, necessarily, the container where the type was registered!)

Confused? I was! In the trivial case where your application only makes use of a single container, there is no difference in behavior between the Hierarchical and ContainerControlled LifetimeManagers. The interesting case arises when you make use of a hierarchy of containers; with the most typical example being that of session management.

## Session Management and Container Scoping

For example, my business user takes calls through my application. I want to be able to isolate my singletons (e.g. service instances, session data, etc.) on a per-call basis, not on a per-application instance basis. In this fashion, I safely isolate my call data from one call to another, but I preserve the utility of resolving singletons out of my container.

Here is a simplified example of the issue at hand:

IUnityContainer unityContainer = new UnityContainer();

unityContainer.RegisterType<ICustomerProfile>(new ContainerControlledLifetimeManager());

I’ve registered my customer profile as a singleton on the root container. Now let’s take a peek inside my application session class:

private readonly IUnityContainer \_scopedContainer;

private readonly ICustomerProfile \_custProfile;

public ApplicationSession(IUnityContainer container)

{

// create a child container, scoped for this session

this.\_scopedContainer = container.CreateChildContainer();

// track a (singleton) ICustomerProfile instance per session

this.\_custProfile = this.\_scopedContainer.Resolve<ICustomerProfile>();

}

public CallData CallData

{

get { return this.\_custProfile.LastCallData; }

set { this.\_custProfile.UpdateCallHistory(value); }

}

Okay, I create a scoped container for the current session, and I resolve an instance of customer profile out of the scoped container. I use the instance of customer profile to manage interactions with the CallData property for the session. Here is where it gets tricky:

// call #1 comes in, I create a session instance

IApplicationSession newSession = unityContainer.Resolve<IApplicationSession>();

sessionManager.AddAndActivateSession(newSession);

newSession.CallData = callData;

// call #2 comes in, I create another session instance

IApplicationSession newSession = unityContainer.Resolve<IApplicationSession>();

sessionManager.AddAndActivateSession(newSession);

newSession.CallData = callData;

Each of my sessions has a distinct scoped child container, but because I used ContainerControlledLifetimeManager when I registered the ICustomerProfile type, both child containers are resolving the ICustomerProfile to the same instance. Therefore, if my user is handling two concurrent calls, he will end up writing customer #2′s call data into customer #1′s call history profile!

Luckily, the only change required to fix this issue is to change the registration of ICustomerProfile from:

unityContainer.RegisterType<ICustomerProfile>(new ContainerControlledLifetimeManager());

to:

unityContainer.RegisterType<ICustomerProfile>(new HierarchicalLifetimeManager());

Now each of my scoped containers will resolve their own instance of the ICustomerProfile singleton, despite the type being originally registered with the root container. (Keep in mind, the above example is grossly simplified in an effort to reduce the explanation to its core issue.)