IoC container usage

Patterns and anti-patterns

Krzysztof Koźmic
http://kozmic.pl

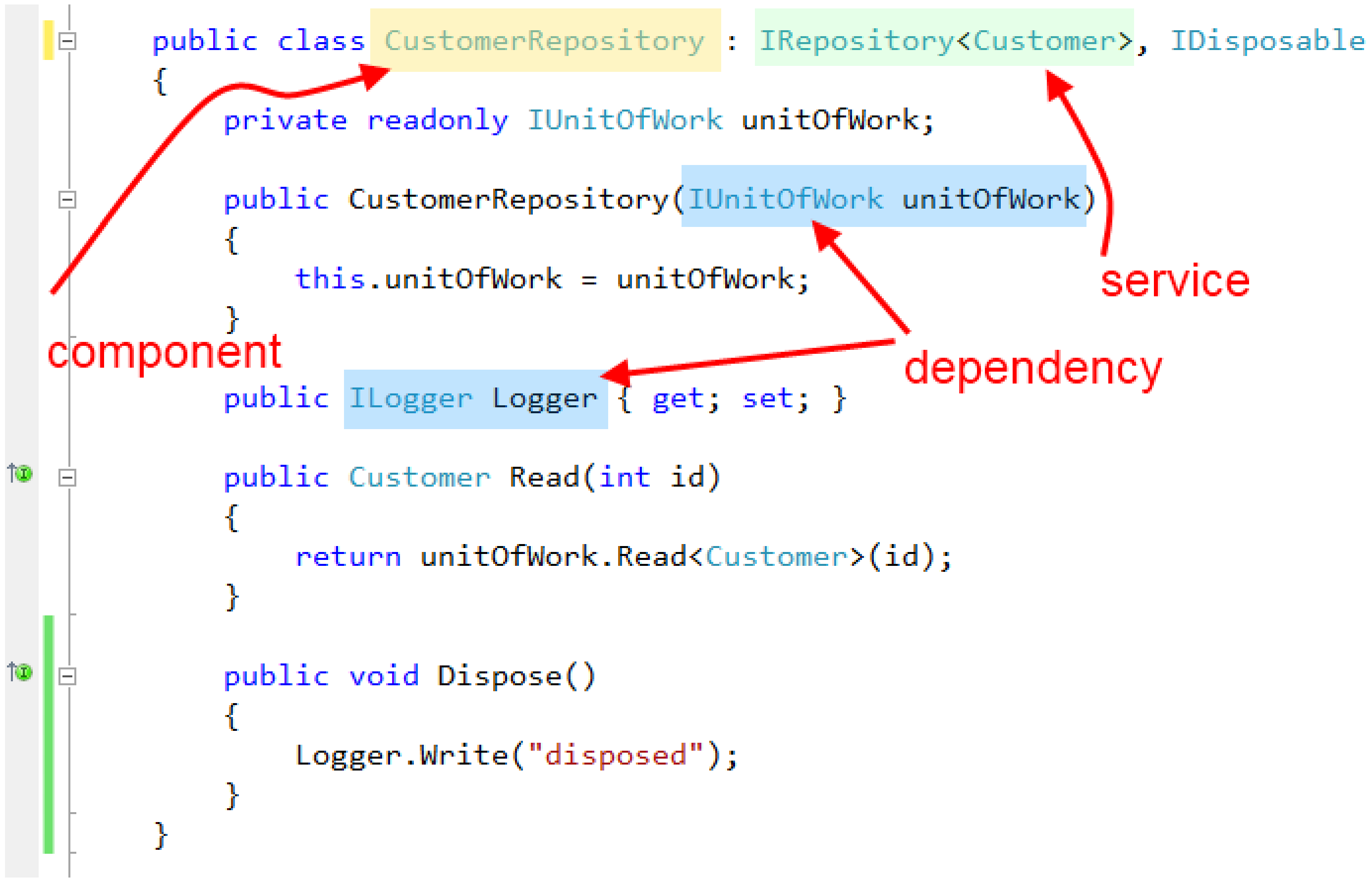
Building applications is hard

Apps are fragile

Components FTW!

Loosely coupled

Component, service, dependency



```
public class CustomerRepository : IRepository<Customer>, IDisposable
{
    private readonly IUnitOfWork unitOfWork;

    public CustomerRepository(IUnitOfWork unitOfWork)
    {
        this.unitOfWork = unitOfWork;
    }

    public ILogger Logger { get; set; }

    public Customer Read(int id)
    {
        return unitOfWork.Read<Customer>(id);
    }

    public void Dispose()
    {
        Logger.Write("disposed");
    }
}
```

Problem – how to manage them?

Using a Container (obviously)!

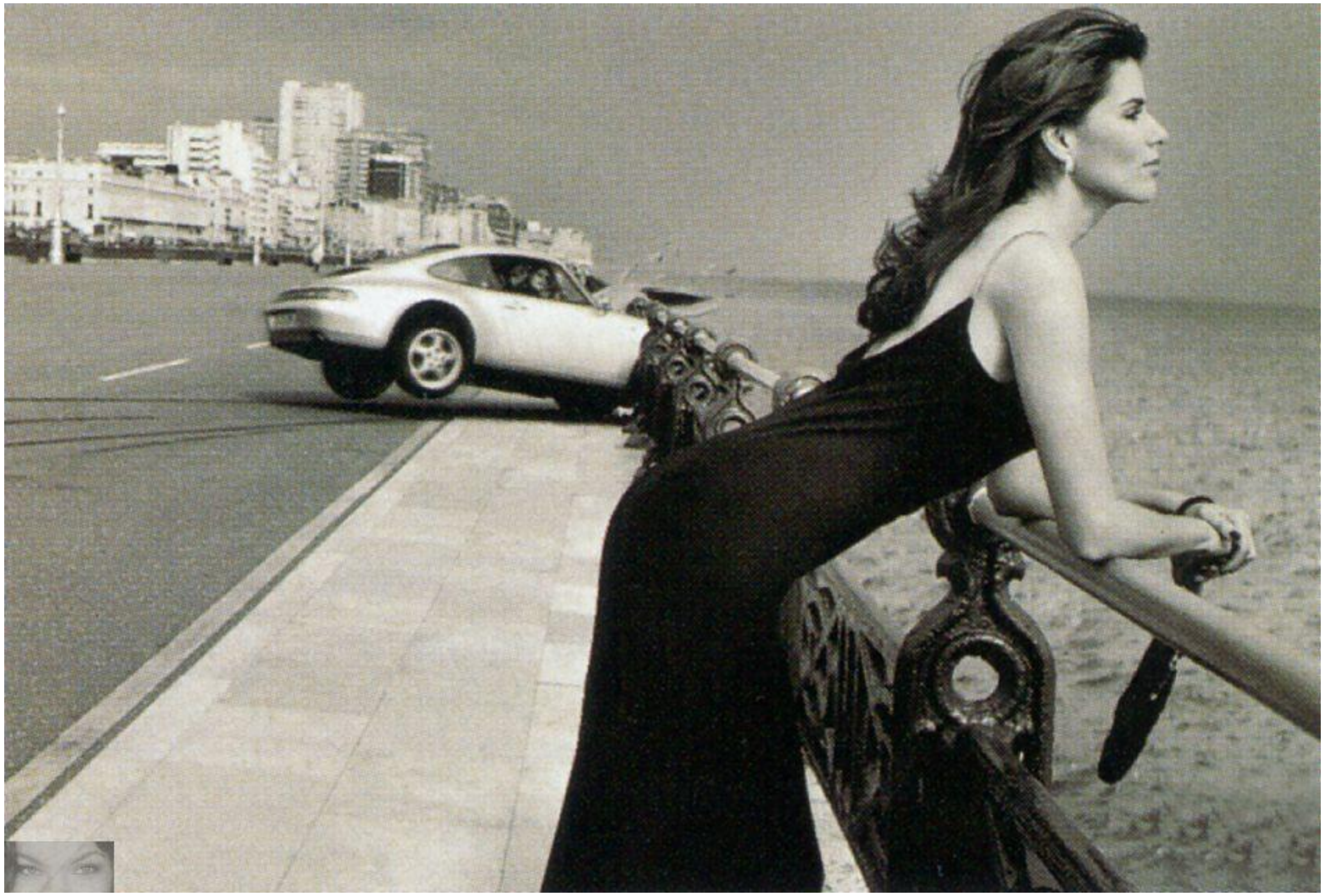
WINDSORCONTAINER

Not always a smooth ride

The “I” word

Misses the point

Distraction



Inversion of Control

The container has you!

All your components are belong to us.

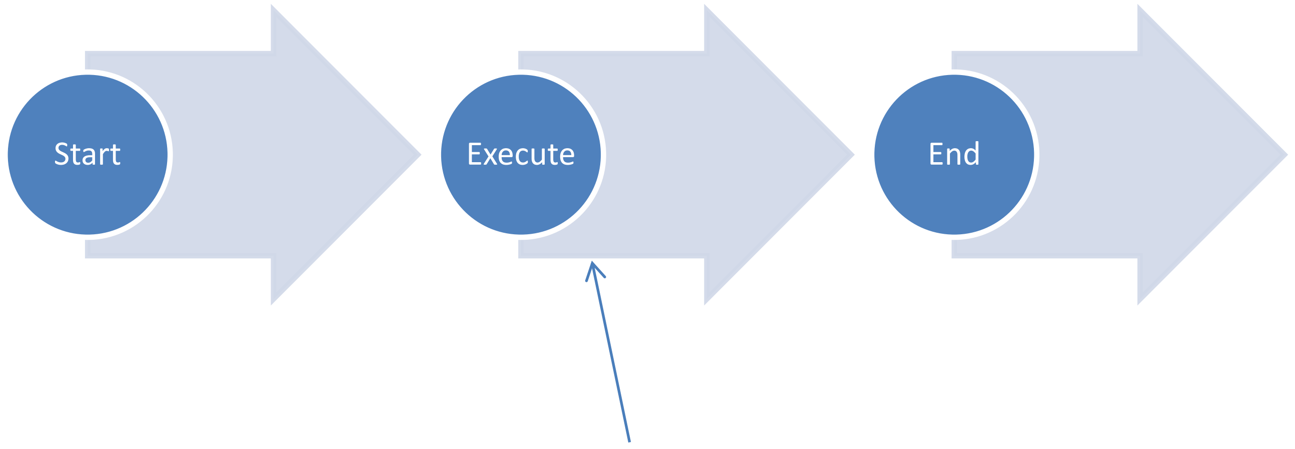
This is important!

again

Container calls down to your app, not the other way around.

How to manage them (again)?

Application lifecycle



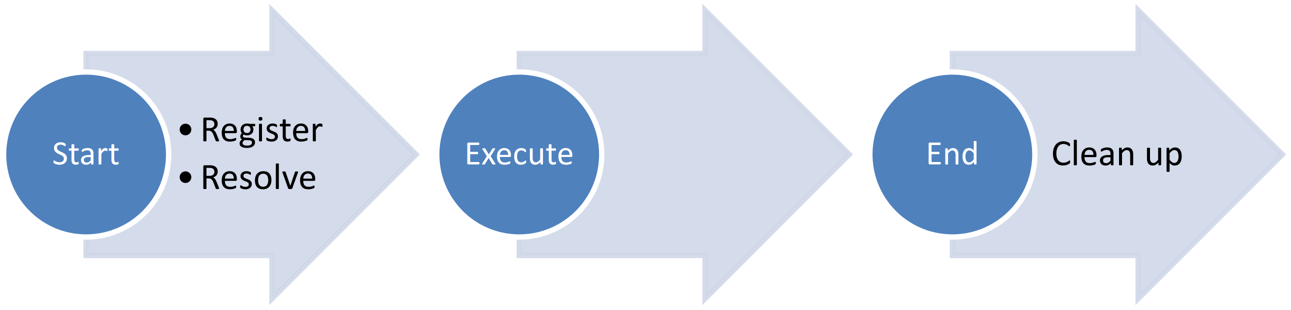
This in reality takes 99% of the time

Example

```
public class GlobalApplication : HttpApplication
{
    public static void OnStart()
    {
        Bootstrapper.Run();
        Log.Info("Application Started");
    }

    public static void OnEnd()
    {
        Log.Warning("Application Ended");
        IoC.Reset();
    }

    protected void Application_Start()
    {
        OnStart();
    }

    protected void Application_End()
    {
        OnEnd();
    }
}
```

plus Controllers

Container lifecycle



The three calls pattern (aka – RRR (Register, Resolve, Release))

Example (application perspective)

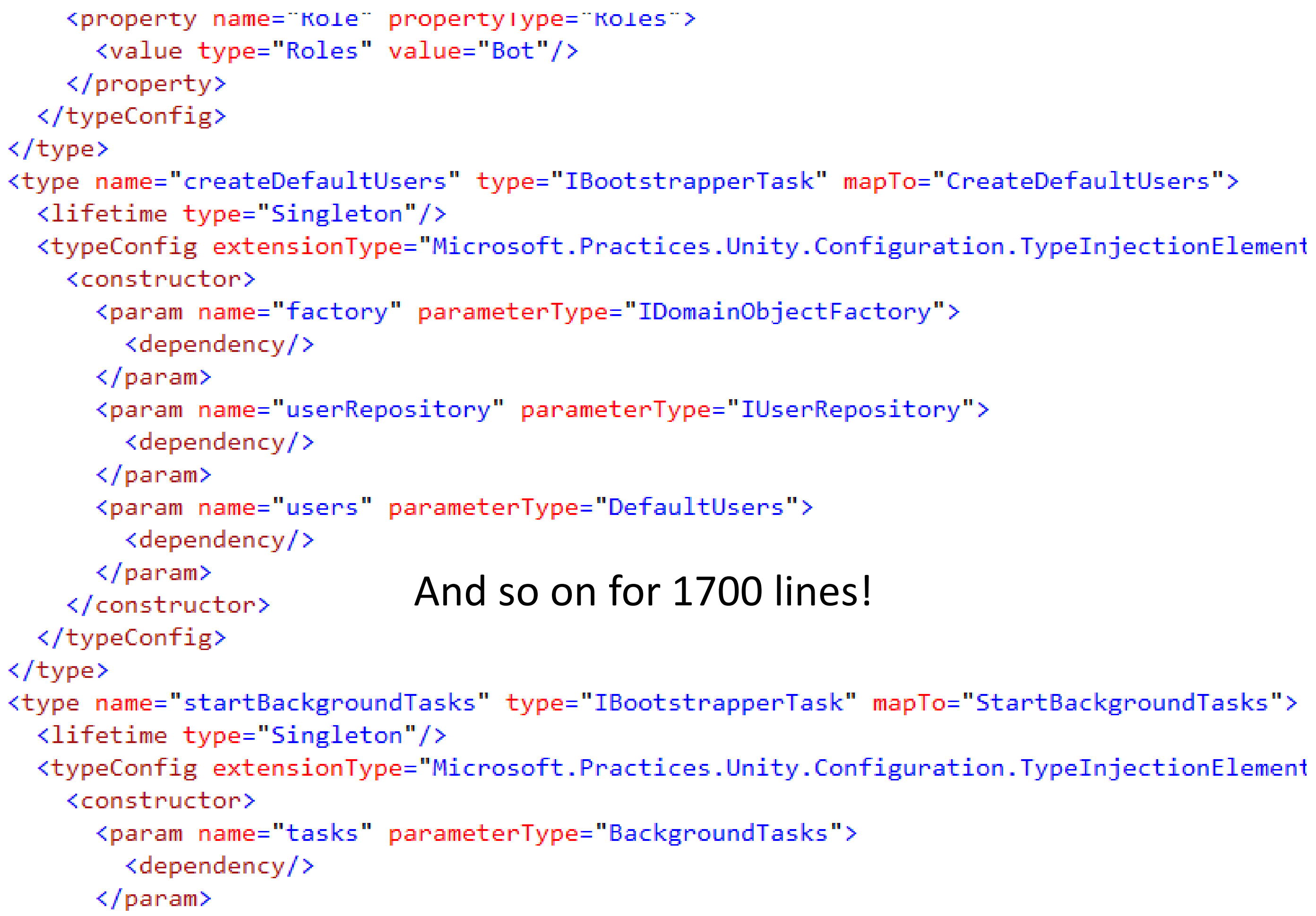
```
public partial class App
{
    private readonly GuyWire guyWire = new GuyWire();

    public App()
    {
        Startup += OnStartup;
        Exit += OnExit;

        InitializeComponent();
    }

    private void OnStartup(object sender, StartupEventArgs e)
    {
        guyWire.Wire();                                register

        RootVisual = guyWire.GetRoot();                resolve
    }

    private void OnExit(object sender, EventArgs e)
    {
        guyWire.Dewire();                              clean up
    }
}
```

Example (digging deeper)

```
public class GuyWire
{
    private readonly IWindsorContainer container;

    public GuyWire()
    {
        container = new WindsorContainer();
    }

    public void Wire()  ←              register
    {
        container.Install(FromAssembly.This());
    }

    public void Dewire()  ←            clean up
    {
        container.Dispose();
    }

    public UIElement GetRoot()  ←      resolve
    {
        return container.Resolve<MainView>();
    }
}
```

Registration (step 1)

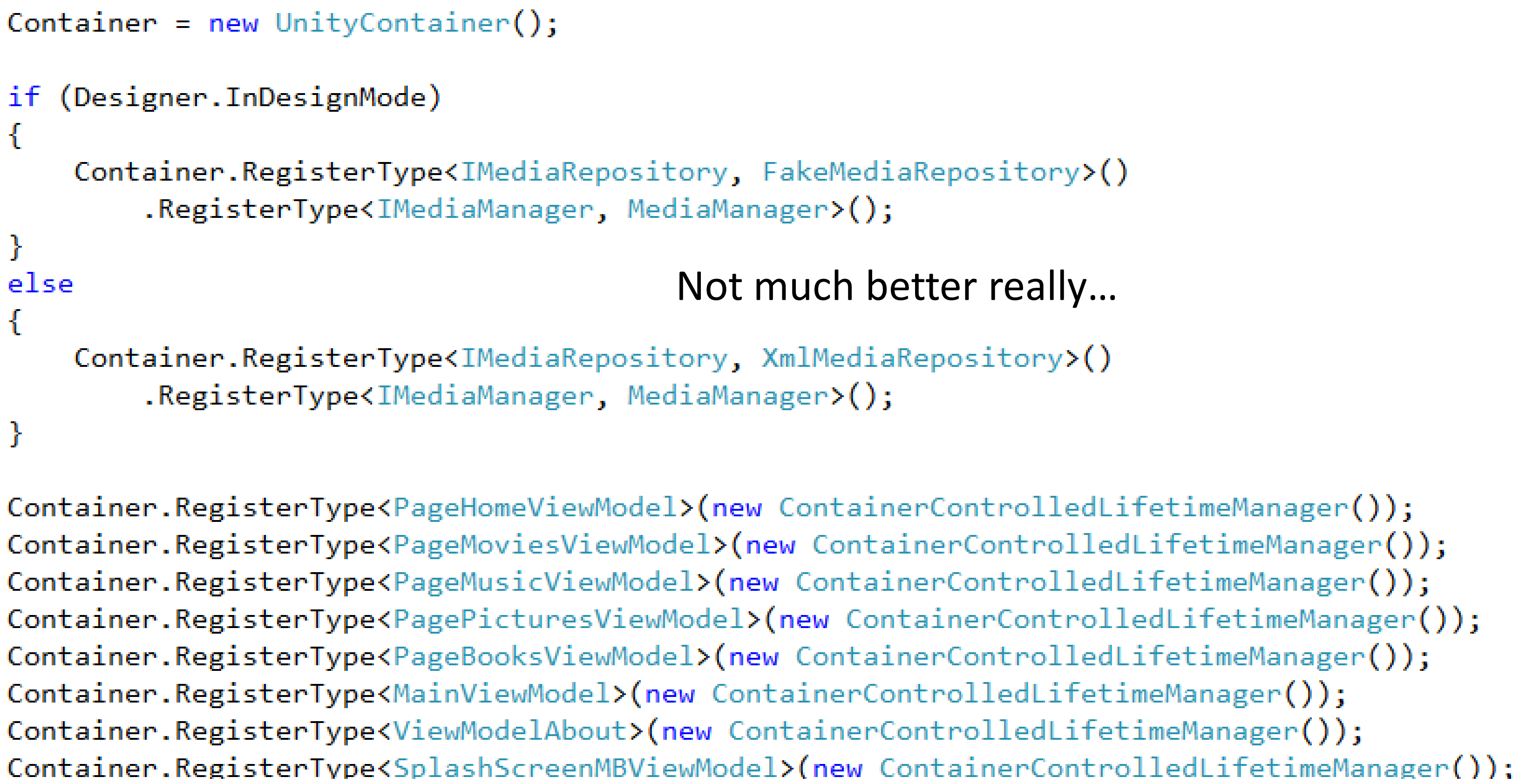
XML (just don’t!)



```
    <property name="Role" propertyType="Roles">
      <value type="Roles" value="Bot"/>
    </property>
  </typeConfig>
</type>
<type name="createDefaultUsers" type="IBootstrapperTask" mapTo="CreateDefaultUsers">
  <lifetime type="Singleton"/>
  <typeConfig extensionType="Microsoft.Practices.Unity.Configuration.TypeInjectionElement
    <constructor>
      <param name="factory" parameterType="IDomainObjectFactory">
        <dependency/>
      </param>
      <param name="userRepository" parameterType="IUserRepository">
        <dependency/>
      </param>
      <param name="users" parameterType="DefaultUsers">
        <dependency/>
      </param>
    </constructor>
  </typeConfig>
</type>
<type name="startBackgroundTasks" type="IBootstrapperTask" mapTo="StartBackgroundTasks">
  <lifetime type="Singleton"/>
  <typeConfig extensionType="Microsoft.Practices.Unity.Configuration.TypeInjectionElement
    <constructor>
      <param name="tasks" parameterType="BackgroundTasks">
        <dependency/>
      </param>
```

And so on for 1700 lines!

Registration in code (control freak)



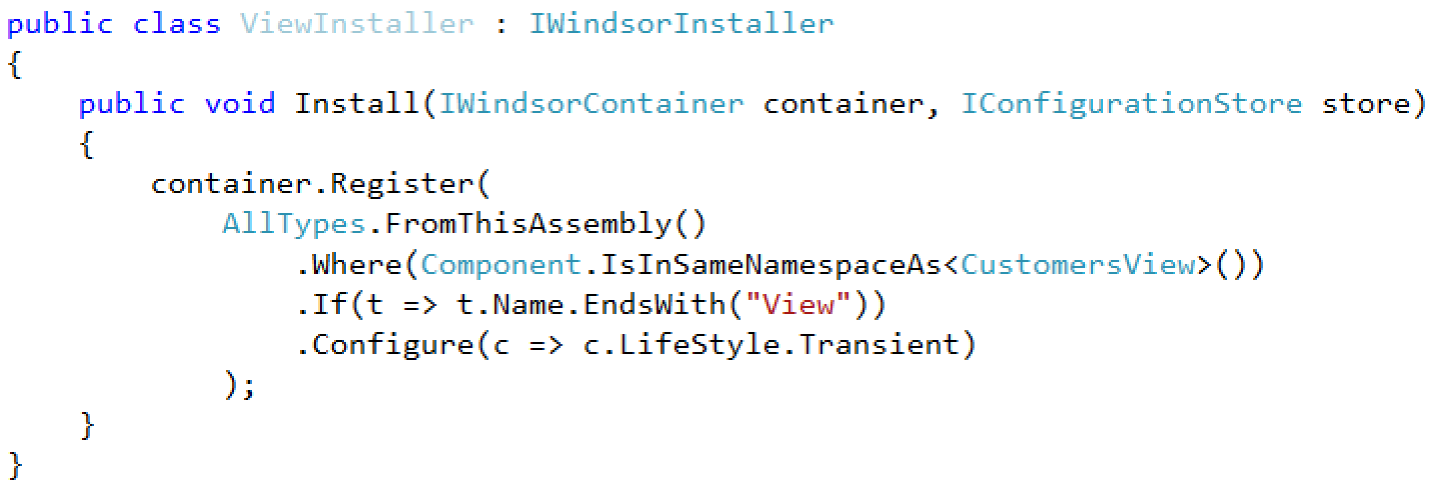
```
Container = new UnityContainer();

if (Designer.InDesignMode)
{
    Container.RegisterType<IMediaRepository, FakeMediaRepository>()
        .RegisterType<IMediaManager, MediaManager>();
}
else
{
    Container.RegisterType<IMediaRepository, XmlMediaRepository>()
        .RegisterType<IMediaManager, MediaManager>();
}

Container.RegisterType<PageHomeViewModel>(new ContainerControlledLifetimeManager());
Container.RegisterType<PageMoviesViewModel>(new ContainerControlledLifetimeManager());
Container.RegisterType<PageMusicViewModel>(new ContainerControlledLifetimeManager());
Container.RegisterType<PagePicturesViewModel>(new ContainerControlledLifetimeManager());
Container.RegisterType<PageBooksViewModel>(new ContainerControlledLifetimeManager());
Container.RegisterType<MainViewModel>(new ContainerControlledLifetimeManager());
Container.RegisterType<ViewModelAbout>(new ContainerControlledLifetimeManager());
Container.RegisterType<SplashScreenMBViewModel>(new ContainerControlledLifetimeManager());
```

Not much better really…

Conventions FTW!

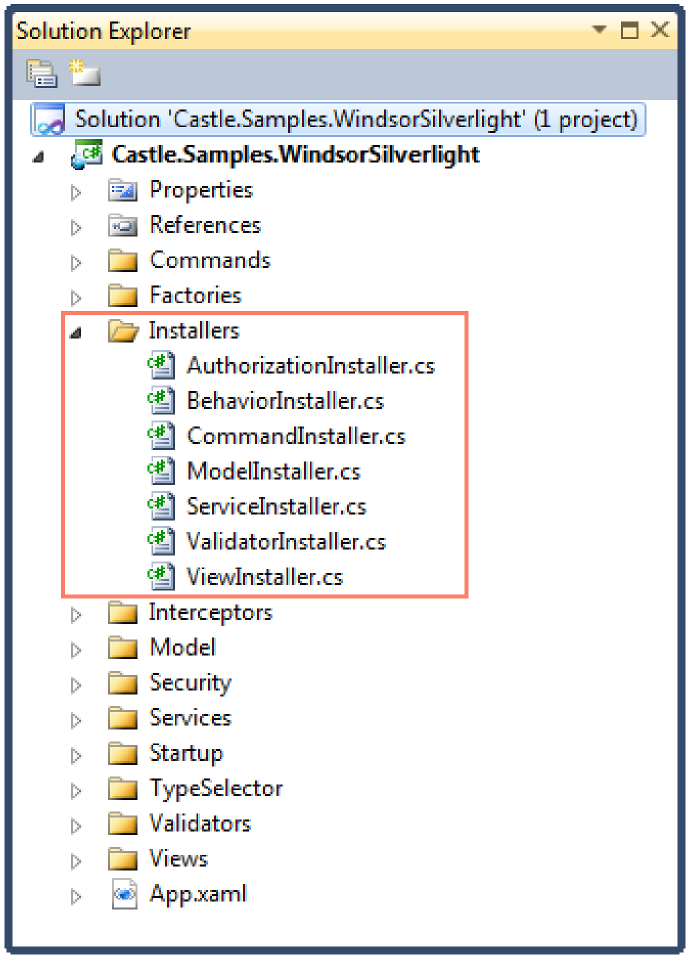
```
public class ViewInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(
            AllTypes.FromThisAssembly()
                .Where(Component.IsInSameNamespaceAs<CustomersView>())
                .If(t => t.Name.EndsWith("View"))
                .Configure(c => c.LifeStyle.Transient)
            );
    }
}
```

Installers (modules/registries)



```csharp
public class ViewInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(
            AllTypes.FromThisAssembly()
                .Where(Component.IsInSameNamespaceAs<CustomersView>())
                .If(t => t.Name.EndsWith("View"))
                .Configure(c => c.LifeStyle.Transient)
            );
    }
}
```

Installers and SRP – keep them small



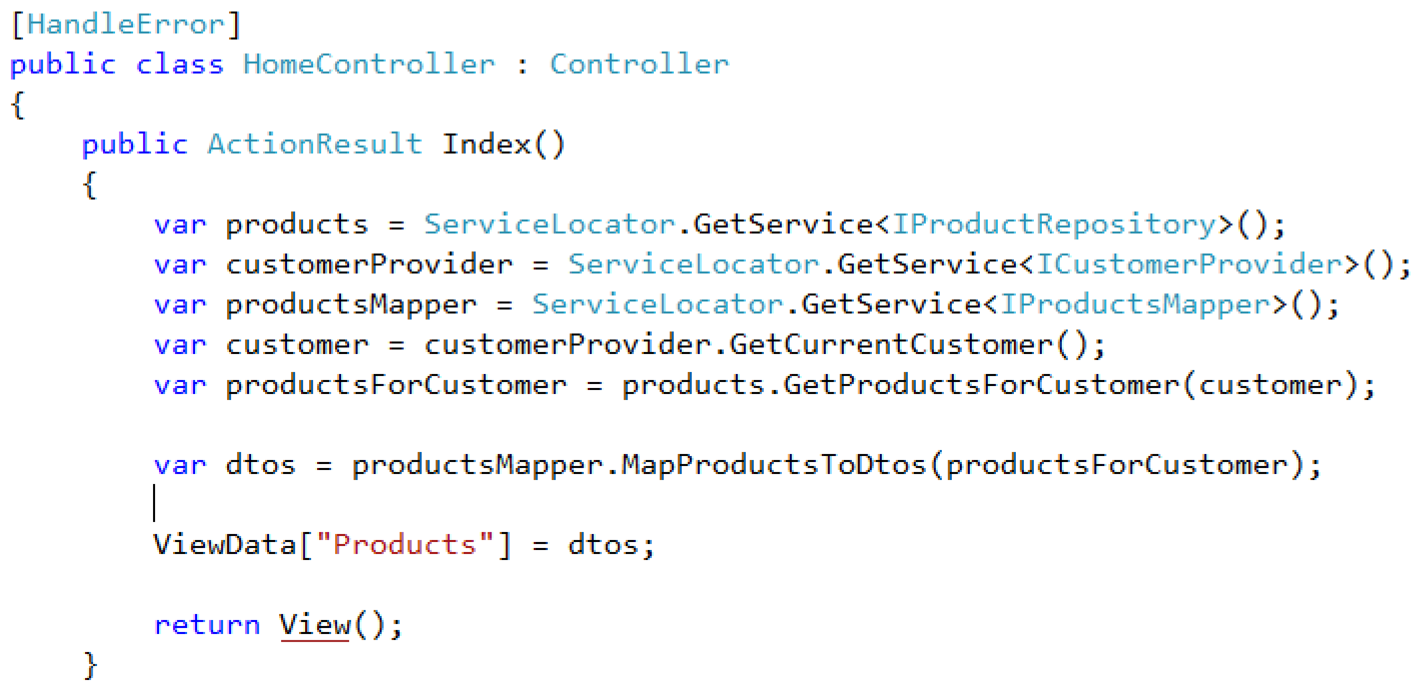
Solution Explorer

Solution 'Castle.Samples.WindsorSilverlight' (1 project)
- Castle.Samples.WindsorSilverlight
  - Properties
  - References
  - Commands
  - Factories
  - Installers
    - AuthorizationInstaller.cs
    - BehaviorInstaller.cs
    - CommandInstaller.cs
    - ModelInstaller.cs
    - ServiceInstaller.cs
    - ValidatorInstaller.cs
    - ViewInstaller.cs
  - Interceptors
  - Model
  - Security
  - Services
  - Startup
  - TypeSelector
  - Validators
  - Views
  - App.xaml

Resolution (step 2)

Only root components

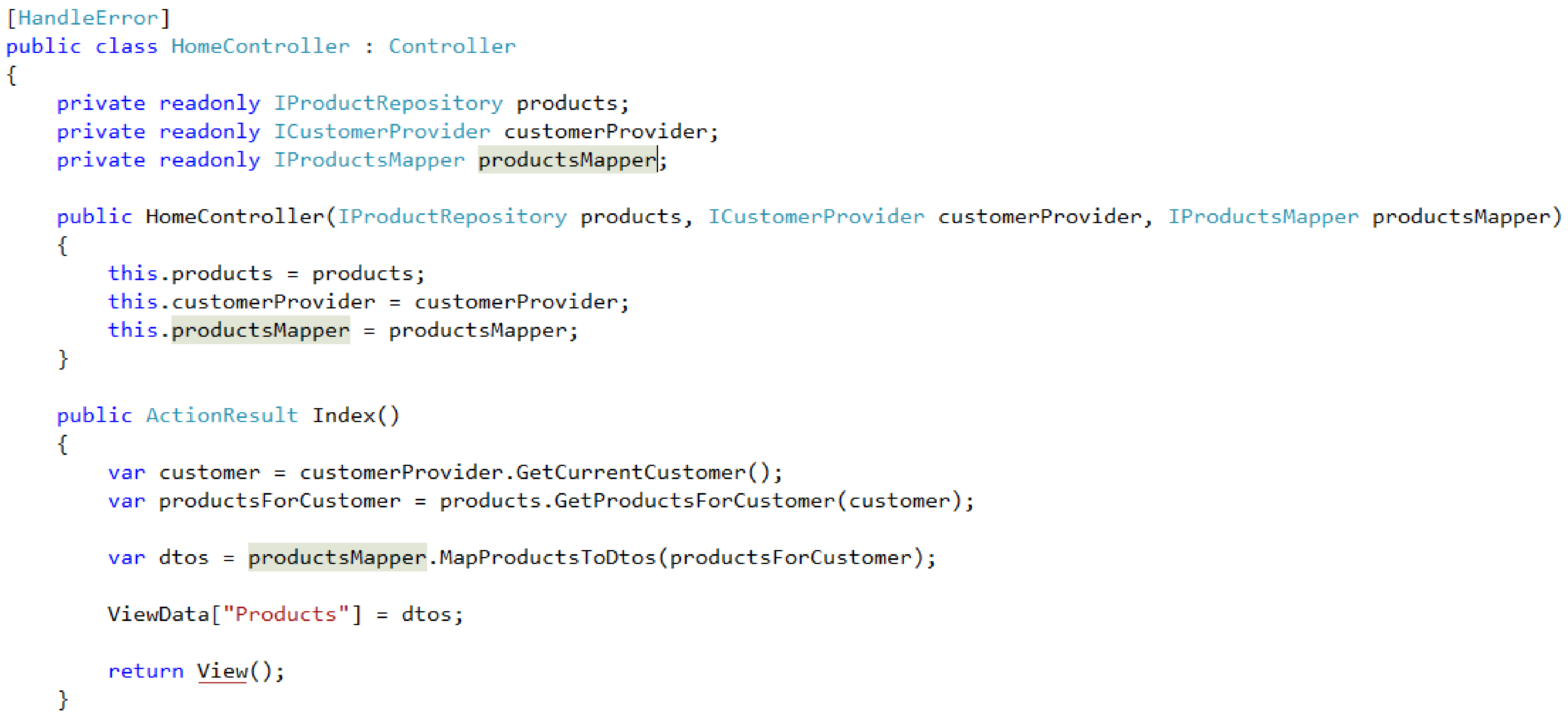
Don’t try this at home (or work)



```
[HandleError]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        var products = ServiceLocator.GetService<IProductRepository>();
        var customerProvider = ServiceLocator.GetService<ICustomerProvider>();
        var productsMapper = ServiceLocator.GetService<IProductsMapper>();
        var customer = customerProvider.GetCurrentCustomer();
        var productsForCustomer = products.GetProductsForCustomer(customer);

        var dtos = productsMapper.MapProductsToDtos(productsForCustomer);

        ViewData["Products"] = dtos;

        return View();
    }
```

Service Locator is teh evil

That’s better



```
[HandleError]
public class HomeController : Controller
{
    private readonly IProductRepository products;
    private readonly ICustomerProvider customerProvider;
    private readonly IProductsMapper productsMapper;

    public HomeController(IProductRepository products, ICustomerProvider customerProvider, IProductsMapper productsMapper)
    {
        this.products = products;
        this.customerProvider = customerProvider;
        this.productsMapper = productsMapper;
    }

    public ActionResult Index()
    {
        var customer = customerProvider.GetCurrentCustomer();
        var productsForCustomer = products.GetProductsForCustomer(customer);

        var dtos = productsMapper.MapProductsToDtos(productsForCustomer);

        ViewData["Products"] = dtos;

        return View();
    }
```

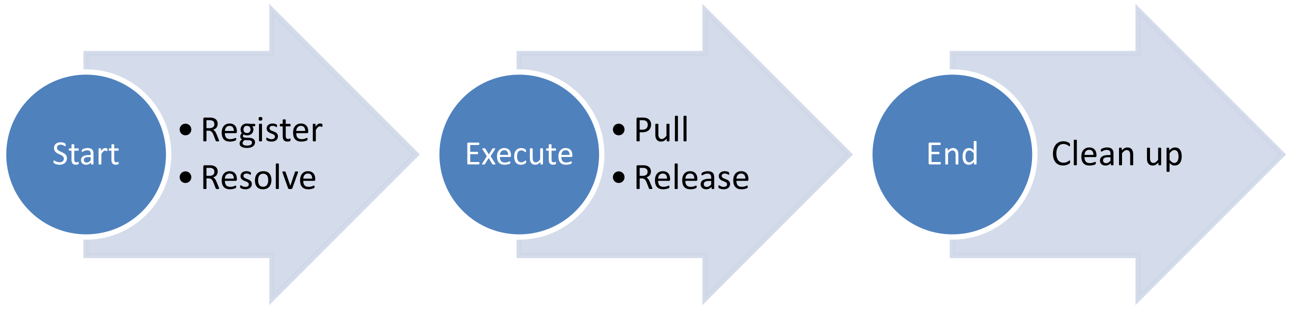
Okay, Houston. Hey, we've got a problem here.

```
public HomeController(IProductRepository products, IProductsMapper productsMapper, IPromotionsFactory promotions,
                      IHandlerFactory factory, ICustomerRepository customers, IOrderRepository orders,
                      IPromotionStrategy promotionStrategy,
                      ICustomerFreeShippingEligibilityCalculator freeShipping, ILogger logger,
                      IProductOfTheMonth productOfTheMonth, ICurrencyConverter currencyConverter,
                      IPaymentProcessor paymentProcessor, IFraudAssesor fraudAssesor)
{
    this.products = products;
    this.fraudAssesor = fraudAssesor;
    this.paymentProcessor = paymentProcessor;
    this.currencyConverter = currencyConverter;
    this.productOfTheMonth = productOfTheMonth;
    this.logger = logger;
    this.freeShipping = freeShipping;
    this.promotionStrategy = promotionStrategy;
    this.orders = orders;
    this.customers = customers;
    this.factory = factory;
    this.promotions = promotions;
    this.productsMapper = productsMapper;
}
```

Static classes

```
public ActionResult Index(string promotionName)
{
    // we can't grab promotion via constructor, because
    // we don't know which one we'll need until this method is called
    var promotion = Promotions.GetPromotion(promotionName);

    var allProducts = products.GetAllProducts();
    promotion.ApplyToAll(allProducts);
    var dtos = productsMapper.MapProductsToDtos(allProducts);

    ViewData["Products"] = dtos;

    return View();
}
```

Container lifecycle (amended)



So how do I pull without referencing the container, huh?

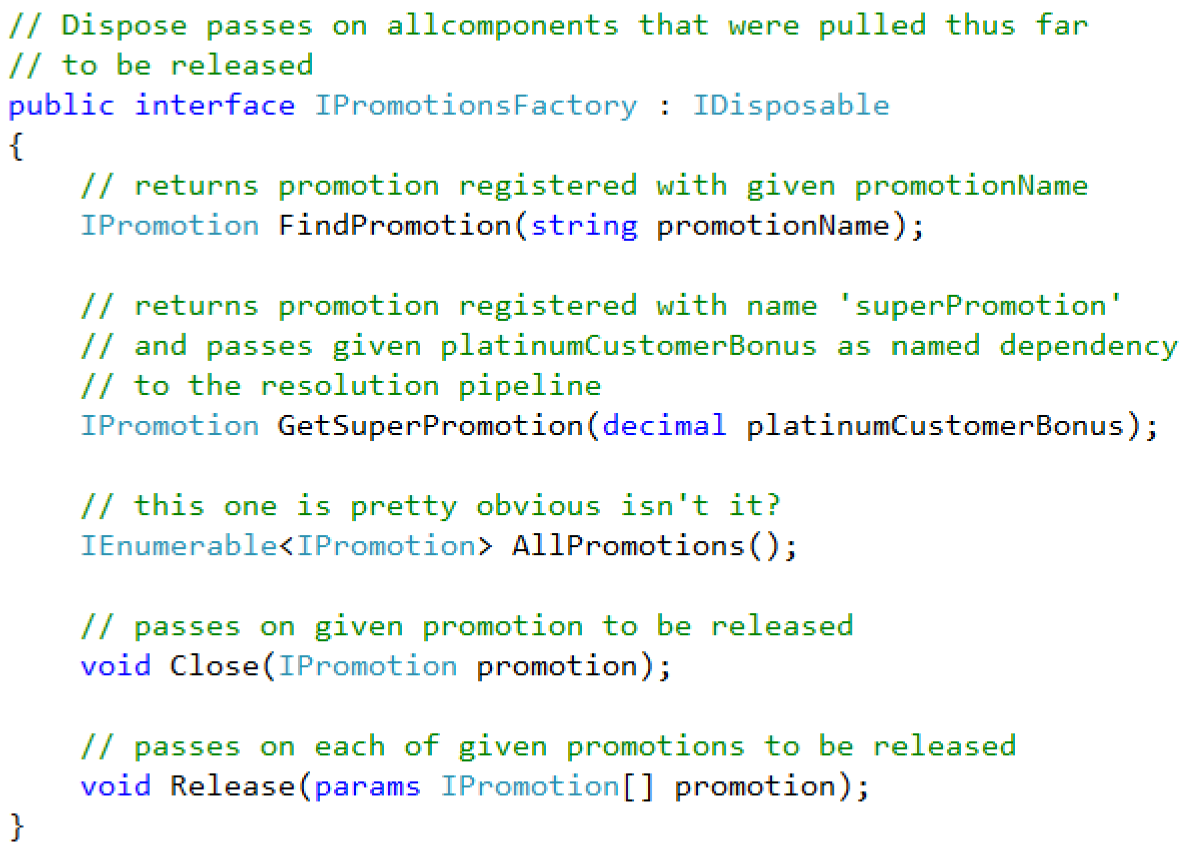
Factory!

```
public HomeController(IProductRepository products, IProductsMapper productsMapper, IPromotionsFactory promotions)
{
    this.products = products;
    this.promotions = promotions;
    this.productsMapper = productsMapper;
}

public ActionResult Index(string promotionName)
{
    // we can't grab promotion via constructor, because
    // we don't know which one we'll need until this method is called
    var promotion = promotions.GetPromotion(promotionName);

    var allProducts = products.GetAllProducts();
    promotion.ApplyToAll(allProducts);
    var dtos = productsMapper.MapProductsToDtos(allProducts);

    ViewData["Products"] = dtos;

    return View();
}
```

Typed Factory

```
public class FactoriesInstaller:IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.AddFacility<TypedFactoryFacility>();
        container.Register(Component.For<IPromotionsFactory>()
                                .LifeStyle.Transient
                                .AsFactory()); // <-- this is important
    }
}
```

Convention over Configuration



```
// Dispose passes on allcomponents that were pulled thus far
// to be released
public interface IPromotionsFactory : IDisposable
{
    // returns promotion registered with given promotionName
    IPromotion FindPromotion(string promotionName);

    // returns promotion registered with name 'superPromotion'
    // and passes given platinumCustomerBonus as named dependency
    // to the resolution pipeline
    IPromotion GetSuperPromotion(decimal platinumCustomerBonus);

    // this one is pretty obvious isn't it?
    IEnumerable<IPromotion> AllPromotions();

    // passes on given promotion to be released
    void Close(IPromotion promotion);

    // passes on each of given promotions to be released
    void Release(params IPromotion[] promotion);
}
```

Those are just defaults – they can be overridden

Clean up (step 3)

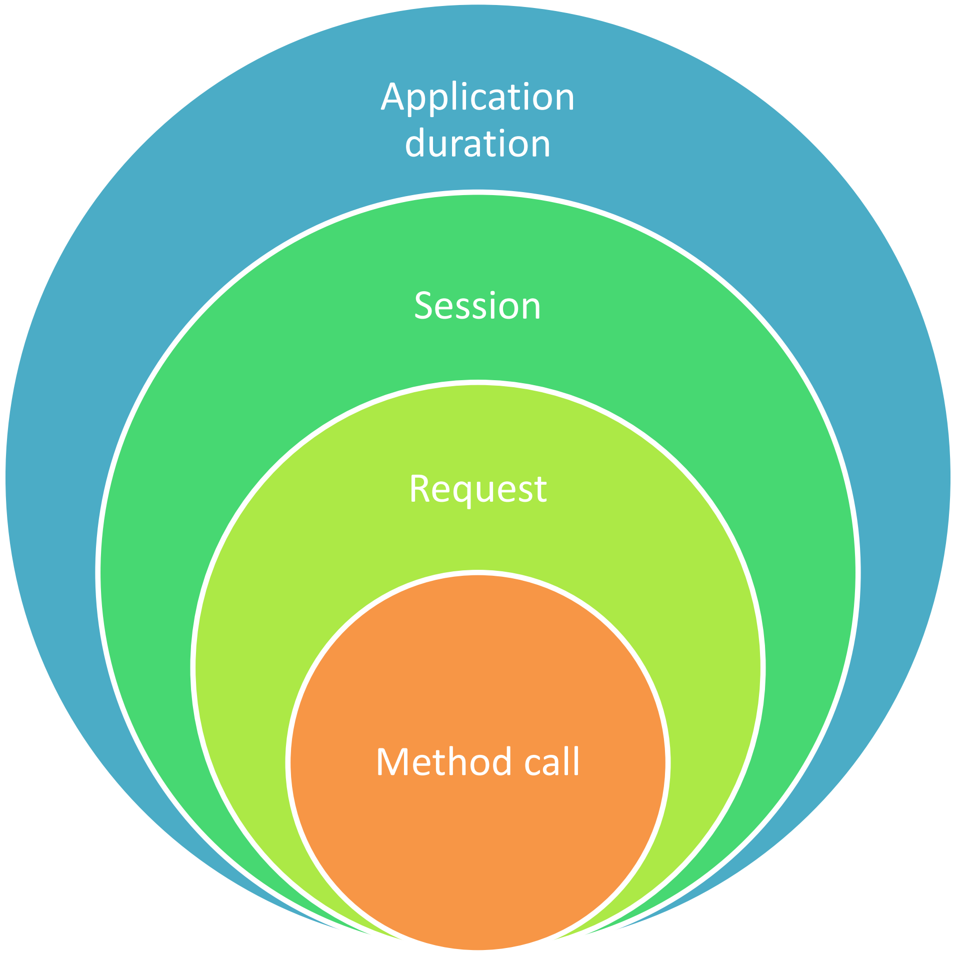
What’s the problem again?

- Who disposes the disposable?
- Who calls Flush/SubmitChanges on Unit Of Work?
- Who unwires wired event handlers?
- Who stops background tasks?
- Etc…

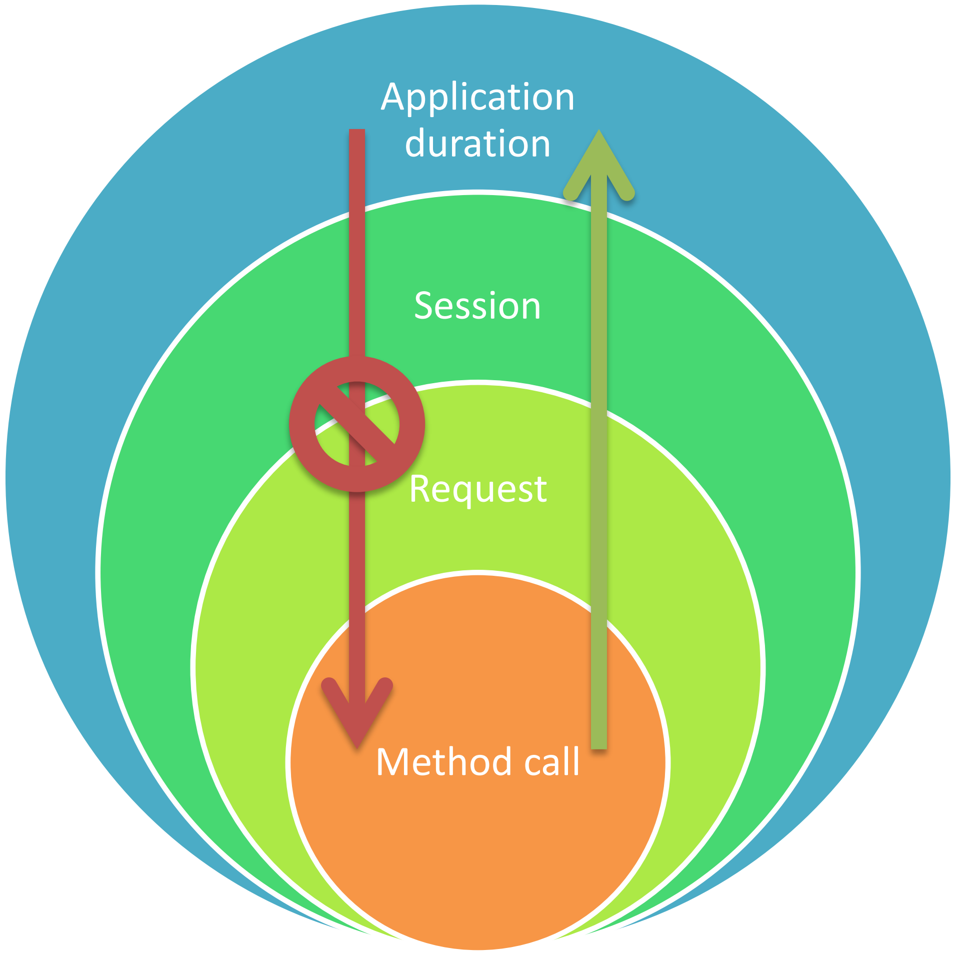
Container owns the components

Container creates objects, and container destroys them.

Lifetime scoping

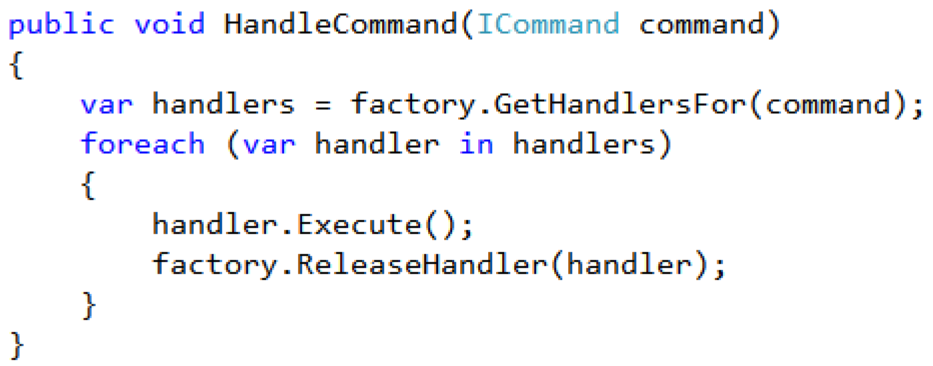


Don’t hold on to components that should live shorter than you



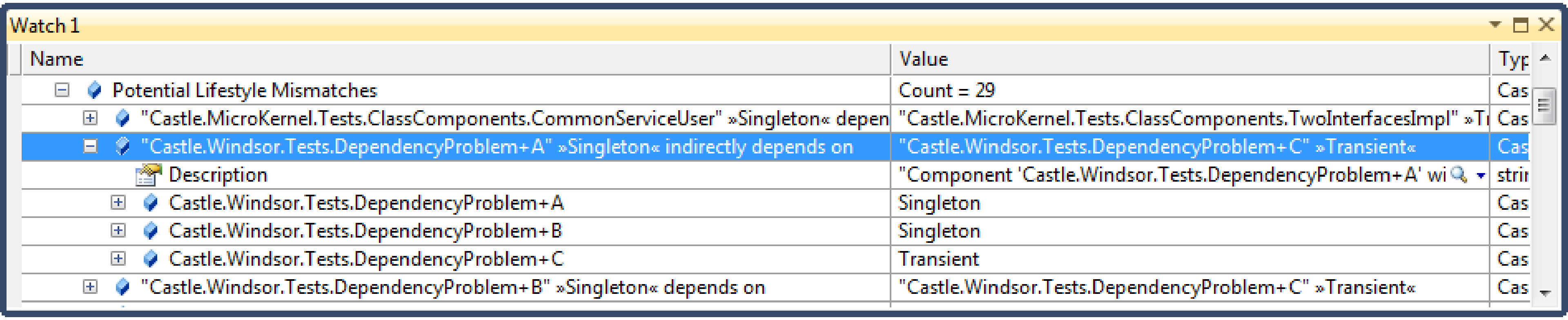
Zombie objects

If you pull them, give them back



```
public void HandleCommand(ICommand command)
{
    var handlers = factory.GetHandlersFor(command);
    foreach (var handler in handlers)
    {
        handler.Execute();
        factory.ReleaseHandler(handler);
    }
}
```

Container can help



Watch 1

Name	Value	Typ
Potential Lifestyle Mismatches	Count = 29	Cas
"Castle.MicroKernel.Tests.ClassComponents.CommonServiceUser" »Singleton« depen	"Castle.MicroKernel.Tests.ClassComponents.TwoInterfacesImpl" »Tı	Cas
"Castle.Windsor.Tests.DependencyProblem+A" »Singleton« indirectly depends on	"Castle.Windsor.Tests.DependencyProblem+C" »Transient«	Cas
Description	"Component 'Castle.Windsor.Tests.DependencyProblem+A' wi	strir
Castle.Windsor.Tests.DependencyProblem+A	Singleton	Cas
Castle.Windsor.Tests.DependencyProblem+B	Singleton	Cas
Castle.Windsor.Tests.DependencyProblem+C	Transient	Cas
"Castle.Windsor.Tests.DependencyProblem+B" »Singleton« depends on	"Castle.Windsor.Tests.DependencyProblem+C" »Transient«	Cas

Container and testing

Validate right types get registered

```
[Test]
public void All_controller_types_are_registered()
{
    var controllerHandlers = container.Kernel.GetAssignableHandlers(typeof(IController));
    var allControllerTypes = new HashSet<Type>(
        typeof (HomeController).Assembly.GetExportedTypes()
            .Where(t => typeof (IController).IsAssignableFrom(t)));
    var registeredControllerTypes = new HashSet<Type>(
        controllerHandlers
            .Select(h => h.ComponentModel.Implementation));

    Assert.IsTrue(allControllerTypes.SetEquals(registeredControllerTypes));
}
```

Validate component's configuration

```
[Test]
public void All_controllers_are_transient()
{
    var controllerHandlers = container.Kernel.GetAssignableHandlers(typeof(IController));
    var nonTransientControllers = controllerHandlers
        .Where(h => h.ComponentModel.LifestyleType != LifestyleType.Transient)
        .ToArray();

    Assert.IsEmpty(nonTransientControllers);
}
```

Validate components are resolvable

```
[Test]
public void All_components_are_resolvable()
{
    var unresolvableComponents = container.Kernel.GetAssignableHandlers(typeof(object))
        .Where(h => h.CurrentState != HandlerState.Valid)
        .Where(h => IsSpecialCase(h) == false)
        .ToArray();

    Assert.IsEmpty(unresolvableComponents);
}
```

Questions?