Table of Contents

[What is Dependency Injection? 2](#_Toc182181656)

[Overview 2](#_Toc182181657)

[Traditional Dependency Management 2](#_Toc182181658)

[The Hard-Wired Service Locator 3](#_Toc182181659)

[The Generic Service Locator 5](#_Toc182181660)

[The Dependency Injection Container 7](#_Toc182181661)

[Using DependencyContainer 8](#_Toc182181662)

[Injection Operations 9](#_Toc182181663)

[Constructor Injection 9](#_Toc182181664)

[Setter Injection 11](#_Toc182181665)

[Method Injection 12](#_Toc182181666)

[Modifying Container Behavior 12](#_Toc182181667)

[Type Mapping 12](#_Toc182181668)

[Singletons vs. Instances 13](#_Toc182181669)

[Event Broker 13](#_Toc182181670)

[Using Event Broker with Code 14](#_Toc182181671)

[Using Event Broker with Attributes 14](#_Toc182181672)

[Method Interception 15](#_Toc182181673)

[Miscellaneous 15](#_Toc182181674)

[Disposing the Container 15](#_Toc182181675)

[Being Aware of Object Lifetime 15](#_Toc182181676)

[Configurators 16](#_Toc182181677)

# What is Dependency Injection?

## Overview

At the 50-thousand foot view, most of software development is about taking large problems and breaking them into smaller and smaller ones, until we’ve identified a bunch of very simple and discrete things to be done. We achieve these simple things by writing *components*, which we piece together to form our applications. The rest is just implementation detail, right? ☺

Let’s take a purposefully simple example of a real world set of components (borrowed from [Jim Weirich’s blog post on dependency injection](http://onestepback.org/index.cgi/Tech/Ruby/DependencyInjectionInRuby.rdoc)):

Authenticator

Stock Quotes

Error Handler

Logger

Database

Web App

Even with this simplified example, you can see that there are several dependencies between each of the components in the system. In the real world, even relatively small applications can end up with hundreds of dependencies, which are traditionally managed by developers.

## Traditional Dependency Management

If you were writing the above application in the traditional style, what might it look like?

public class WebApp  
{  
    readonly Authenticator authenticator;  
    readonly Database database;  
    readonly ErrorHandler errorHandler;  
    readonly Logger logger;  
    readonly StockQuotes stockQuotes;

    public WebApp()  
    {  
        authenticator = new Authenticator();  
        database = new Database();  
        errorHandler = new ErrorHandler();  
        logger = new Logger();

        stockQuotes = new StockQuotes();

    }  
}

Many applications are built exactly like this: create and use an object when you need it. For applications with a small number of dependencies, creating the objects as you need them might well be the best strategy.

What happens when the applications start to grow? You may not realize it, but there are a number of implications in that handful of code there: decisions were made, explicitly or not, about the way the system works.

What are the assumptions we’ve made?

First, none of those constructors take parameters. They’re obviously doing the same thing: creating objects in their constructor. In our example, how does the StockQuotes object find the Logger? How would all the classes share a logger instance? You have to make each Logger client stop using new in favor of some kind of singleton pattern instead (turn Logger into a static class, or use a static instance property to get the single instance of the logger).

Second, what about testing? When we want to isolate these components from one another, how do we do that? The only components that appear to be isolated from the world are the Database and ErrorHandler. When we need to test the Logger, how do we put in a stubbed out version of error handling? How do I use a mock database so I don’t need a real one?

Third, what about substitution? When I need to switch from one logger implementation to another, I end up changing calls to multiple places where I’ve hard-coded my dependency on the Logger class. We call these *tightly coupled* components, because WebApp knows exactly which class is doing the logging.

These questions and many more often lead people to consider alternatives to simply creating objects as needed. One common solution that people choose is the *Service Locator*.

## The Hard-Wired Service Locator

We can introduce a hard-wired service locator into our application, which might have an interface like this:

public interface IServiceLocator

{

    Authenticator Authenticator { get; }

    Database Database { get; }

    ErrorHandler ErrorHandler { get; }

    Logger Logger { get; }

    StockQuotes StockQuotes { get; }

}

Now we get to make some decisions about how to implement the locator. For example, we asked earlier how we’d be able to handle singletons. This implementation makes everything a singleton:

public class SingletonServiceLocator : IServiceLocator

{

    readonly Authenticator authenticator;

    readonly Database database;

    readonly ErrorHandler errorHandler;

    readonly Logger logger;

    readonly StockQuotes stockQuotes;

    public ServiceLocator()

    {

        errorHandler = new ErrorHandler();

        database = new Database();

        logger = new Logger(this);

        authenticator = new Authenticator(this);

        stockQuotes = new StockQuotes(this);

    }

    public Authenticator Authenticator { get { return authenticator; } }

    public Database Database { get { return database; } }

    public ErrorHandler ErrorHandler  { get { return errorHandler; } }

    public Logger Logger { get { return logger; } }

    public StockQuotes StockQuotes   { get { return stockQuotes; } }

}

When multiple users ask for an object, they all get the same instance. What if we decided that everybody who asks for a logger needs to get a new one? Now we have a centralized place to make that change, and none of the consumers of the logger need to be concerned about this decision.

Notice that some of the objects we created required us to pass in a service locator, and some don’t. If you look back at our original dependency diagram, there were two classes which had no dependencies: Database and ErrorHandler. This means that when we create the Logger object, we need to know what dependencies it relies upon, so we can ensure that they’re already available. This is why the order of the constructor here looks different than WebApp: we’re creating the components in a “bottom up” fashion, ensuring that lower level dependencies are available before higher level things get created.

We could eliminate this problem by delaying the creation of objects until they’re asked for:

public class DelayCreationSingletonServiceLocator : IServiceLocator

{

Authenticator authenticator;

Database database;

ErrorHandler errorHandler;

Logger logger;

StockQuotes stockQuotes;

public Authenticator Authenticator

{

get

{

if (authenticator == null) authenticator = new Authenticator(this);

return authenticator;

}

}

public Database Database

{

get

{

if (database == null) database = new Database();

return database;

}

}

public ErrorHandler ErrorHandler

{

get

{

if (errorHandler == null) errorHandler = new ErrorHandler();

return errorHandler;

}

}

public Logger Logger

{

get

{

if (logger == null) logger = new Logger(this);

return logger;

}

}

public StockQuotes StockQuotes

{

get

{

if (stockQuotes == null) stockQuotes = new StockQuotes(this);

return stockQuotes;

}

}

}

By using delayed creation, we were able to avoid the problem of ordering entirely. Now if someone asks for the Authenticator component, it doesn’t really matter whether we’ve made its dependencies yet; during the constructor it asks the service locator for its dependencies:

public class Authenticator

{

readonly ErrorHandler errorHandler;

readonly Logger logger;

readonly Database database;

public Authenticator(IServiceLocator locator)

{

errorHandler = locator.ErrorHandler;

logger = locator.Logger;

database = locator.Database;

}

}

Now we’ve solved two of the problems in our original list: easily change lifetime decisions (singletons vs. instances), and creation order. We can solve the final one (substitutability) by use interfaces/abstract classes rather than the actual implementation types. This is as simple as changing the service locator interface to return interface types instead of concrete types:

public interface IServiceLocator

{

IAuthenticator Authenticator { get; }

IDatabase Database { get; }

IErrorHandler ErrorHandler { get; }

ILogger Logger { get; }

IStockQuotes StockQuotes { get; }

}

The subtitutability problem helps us a lot when it’s time to test. Now that we have interfaces instead of concrete types, it’s trivial for us to write an implementation of IServiceLocator that returns dummy objects (mocks, stubs, spies, etc.) for testing purposes. Now our components are *loosely coupled* because they don’t need to know any of the implementation details of which concrete class they’re using.

It looks like we’ve solved all of our original problems; however, we’ve introduced a few more along the way:

* The service locator interface constantly needs to change as we discover new components that are needed in dependencies. Not only is this a maintenance burden, it also makes it more difficult to bring together components that weren’t compiled at the same time (since they have different versions of the service locator interface).
* It’s difficult to make per-request decisions about how to service the request. For instance, there’s no way for us to say “everybody shares the same logger, except for the StockQuotes module, who for security reasons needs to use a different instance (and perhaps a different implementation)”.
* The implementation of the service locator is very specific to your current needs. You can’t easily reuse this class later in another application, since all the logic is hard-coded.

## The Generic Service Locator

By turning the service locator into a generic facility, we can resolve some of the issues above.

public interface IServiceLocator

{

TService Get<TService>();

}

Using this interface is relatively simple: call Get<T> for whatever type of service you need, and if it can find it, it will return it for you. From the consumer point of view, the code looks similar to before:

public class Authenticator : IAuthenticator

{

readonly IDatabase database;

readonly IErrorHandler errorHandler;

readonly ILogger logger;

public Authenticator(IServiceLocator locator)

{

errorHandler = locator.Get<IErrorHandler>();

logger = locator.Get<ILogger>();

database = locator.Get<IDatabase>();

}

}

Instead of having access to the component directly through a property, we now call Get with the appropriate type. While we could implement the service locator with a giant bunch of if statements checking for types and returning our instances, we would prefer a more generalized version.

The interface we defined for IServiceLocator only has a Get method, which seems right, since the *users* of the locator are really looking to find services. However, the code that *creates and populates* the service locator the first time is going to need some way to do that.

To help them do this, we’ll add a Register() method to the locator which can be used to register service instances. While we’ve decided to leave this off the interface, you may need it to be there if your locator users will also need to register their own services.

Our first generic implementation ends up being quite simple:

public class ServiceLocator : IServiceLocator

{

readonly Dictionary<Type, object> services = new Dictionary<Type, object>();

public TService Get<TService>()

{

object result;

if (services.TryGetValue(typeof(TService), out result))

return (TService)result;

throw new ArgumentException("Unknown service type " + typeof(TService).FullName);

}

public void Register<TService>(TService serviceInstance)

{

services[typeof(TService)] = serviceInstance;

}

}

Now that we have our generalized implementation, here is how we might initialize it:

ServiceLocator locator = new ServiceLocator();

locator.Register<IErrorHandler>(new ErrorHandler());

locator.Register<IDatabase>(new Database());

locator.Register<ILogger>(new Logger(locator));

locator.Register<IAuthenticator>(new Authenticator(locator));

locator.Register<IStockQuotes>(new StockQuotes(locator));

Now we have a fully initialized locator that anyone who needs access to any of the services can use.

Notice that we’re back to having ordering being important (the order that items are added to the locator is identical to our original hard-wired service locator). Unlike the hard-wired case, though, we don’t really have any easy tricks to avoid this ordering problem.

Why can’t we just let the service locator create the objects? The reason is simple: some of them require constructor parameters. How is the service locator supposed to know how to pass parameters to the constructor? For that matter, if there are multiple constructors, how does it know which ones to call? Additionally, we’ve lost the ability to say “this thing is a singleton, but this thing is not” because we can’t create the object when it’s requested.

And there are other problems still buried here. For starters, when I have a service locator, how do I know what kinds of objects I can ask for? It can’t simple create anything I ask for, so I have to know which questions it can answer. This wasn’t a problem with our hard-wired service locator.

Secondly, while we’ve loosely coupled the services from each other, we’ve tightly coupled them to the service locator. **They can’t operate without one.** Even more worrisome is how you determine exactly which things a component needs from the locator. Looking at its public constructor, all you can see if a reference to the locator. If you’re writing the locator, how do you know what services to put inside of it to make sure that each component has its requirements satisfied?

Finally, we haven’t really talked yet about what to do about non-services. Ideally the container should be able to provide named primitive values as well as typed services.

We could continue evolving the features of our service locator, introducing things like factories, reflection, etc., but it’s clear that we could probably use a design change to solve some of these problems.

Enter Dependency Injection.

## The Dependency Injection Container

Previously, we had an interface to the service locator (IServiceLocator) as well as various implementations of that interface. That separation was really about the code that *used* the locator (via the interface) vs. the code that *created and populated* the locator (via the concrete type).

With a dependency injection container, we eliminate the interface and the dependency on it. Let’s re-examine our Authenticator component re-written to be used with dependency injection:

public class Authenticator : IAuthenticator

{

readonly IDatabase database;

readonly IErrorHandler errorHandler;

readonly ILogger logger;

public Authenticator(IDatabase database, IErrorHandler errorHandler, ILogger logger)

{

this.database = database;

this.errorHandler = errorHandler;

this.logger = logger;

}

}

Notice that there is no mention of locators or containers; instead, we use the constructor to express our dependencies explicitly, and assume that someone will give those dependencies to us.

You can think of the dependency injection container as a very smart version of “new” which tries to fill in all the constructor parameters for you. Where you would’ve considered calling “new”, you can instead call Get on the container; or, even better, simply ask for your dependencies in the constructor and ignore the container altogether. Don’t think about how your dependencies are fulfilled: let someone else provide them for you, and just use them.

This is very advantageous. When we write Authenticator, we only need to think about the other components we’re going to need access to, not about the locator or the container. Code which wants to use the Authenticator outside the confines of a container or locator can do so, because it can quite easily see the required dependencies.

From a usability stand-point, both as the author of Authenticator and the consumer of Authenticator, this seems like a dramatic improvement over the locator. The dependencies are spelled out quite explicitly.

What does our container API look like now? Here’s a starting point:

public interface IDependencyContainer

{

TService Get<TService>();

void Map<TFrom, TTo>() where TTo : TFrom;

}

Like the old container, we still have a Get() method to retrieve something from the container. We’ve also added a Map() method to allow the code that’s creating the container to configure it properly:

DependencyContainer container = new DependencyContainer();

container.Map<IAuthenticator, Authenticator>();

container.Map<IDatabase, Database>();

container.Map<IErrorHandler, ErrorHandler>();

container.Map<ILogger, Logger>();

container.Map<IStockQuotes, StockQuotes>();

The purpose of the Map() method is to say “if someone needs an IFoo, what you should really build is a Foo”. Notice that our ordering issues are gone now, because we’re not actually creating the objects right now. We’re delaying the creation, but in a way that’s different from our hard-wired service locator (implementation will follow).

Generally speaking, there are going to be very few direct consumers of the container. Often, the main entry point of the application will initialize the container and then get a “root” object out of it which will begin the chain reaction of object creation. Most consumers will be indirect consumers, like Authenticator is. (If you’re using an application framework, often that root object will be something like an Application object.)

Whenever you feel the need to become a direct consumer (i.e., your consumer takes IDependencyContainer), step back and ask yourself if there’s a clearer way to accomplish what you want.

If you don’t know ahead of time which objects (or how many) you will need, you’ll probably need to become a direct consumer of the container. Usually, though, even in those situations, it’s best to create a service you can rely upon instead (say, a FooFactory for creating Foo objects), because that better describes your actual usage, and is easier than a generic container to provide in test scenarios.

Enough talking about containers in the abstract. Let’s look at DependencyContainer, a sample from the CodePlex team.

# Using DependencyContainer

The DependencyContainer class is a sample container which behaves like a service locator, using dependency injection techniques. You can find this sample in the [source code](http://www.codeplex.com/ObjectBuilder/SourceControl/ListDownloadableCommits.aspx) for [ObjectBuilder project](http://www.codeplex.com/ObjectBuilder) on CodePlex, under /Samples/CodePlexContainer.

We’ll examine each feature of the container, talking about how the container behaves by default for that feature, and how you can influence that behavior with code, metadata, and even external configuration like XML.

## Injection Operations

DependencyContainer supports three core forms of injection: *constructor injection*, *setter injection*, and *method injection*.

### Constructor Injection

*Constructor Injection* is the process whereby the container chooses which constructor to use to create an object and how to satisfy the dependencies expressed there.

The default behavior of the container is that it can create an object without being told ahead of time how to build it. For example, here we build an instance of the built-in Object type in .NET:

[Fact]

public void CanCreateObject()

{

DependencyContainer container = new DependencyContainer();

object result = container.Get<object>();

Assert.NotNull(result);

}

In its default configuration, the container can also be used to recursively make an object and its dependencies:

class MyDependencyClass

{

public readonly object obj;

public MyDependencyClass(object obj)

{

this.obj = obj;

}

}

[Fact]

public void CanCreateObjectAndItsDependencies()

{

DependencyContainer container = new DependencyContainer();

MyDependencyClass result = container.Get<MyDependencyClass>();

Assert.NotNull(result);

Assert.NotNull(result.obj);

}

Even though the container has not been taught anything about MyDependencyClass, it’s not only capable of making that class but also of fulfilling its dependency on an instance of the “object” class.

What happens if the object I’m asking to create has two constructors?

class MyDoubleDependencyClass

{

public MyDoubleDependencyClass(object obj)

{

Console.WriteLine("In Object constructor");

}

public MyDoubleDependencyClass(DateTime date)

{

Console.WriteLine("In DateTime constructor");

}

}

[Fact]

public void CanCreateObjectWithTwoConstructors()

{

DependencyContainer container = new DependencyContainer();

MyDoubleDependencyClass result = container.Get<MyDoubleDependencyClass>();

Assert.NotNull(result);

}

If I run this test, I get the following output:

In Object constructor

In an attempt to do its best effort to create an object, the container selected one of the constructors at (not really) random and used that to constructor the object. (I say “not really” because in reality it takes the first constructor returned from reflection, whose order is supposed to be undefined.)

You can influence which constructor is selected for injection by using the [InjectionConstructor] attribute from the ObjectBuilder namespace, like this:

class MyDoubleDependencyClass

{

public MyDoubleDependencyClass(object obj)

{

Console.WriteLine("In Object constructor");

}

[InjectionConstructor]

public MyDoubleDependencyClass(DateTime date)

{

Console.WriteLine("In DateTime constructor");

}

}

[Fact]

public void CanCreateObjectWithTwoConstructors()

{

DependencyContainer container = new DependencyContainer();

MyDoubleDependencyClass result = container.Get<MyDoubleDependencyClass>();

Assert.NotNull(result);

}

Now when I run this test, I get this output:

In DateTime constructor

What if you don’t want the container to create the object for you? Instead of calling Get, there is a method named Inject which can be used to perform dependency injection on an existing object. Calling Inject does all forms of injection *except for constructor injection*, since the object already exists.

### Setter Injection

*Setter Injection* is the process where the container can automatically set dependencies by setting properties on the object.

By default, the container will not set properties. For example:

class MyPropertyDependencyClass

{

object objValue;

public object ObjValue

{

get { return objValue; }

set { objValue = value; }

}

}

[Fact]

public void CanSetProperties()

{

DependencyContainer container = new DependencyContainer();

MyPropertyDependencyClass result = container.Get<MyPropertyDependencyClass>();

Assert.Null(result.ObjValue);

}

After the container has created MyPropertyDependencyClass for us, we can see that ObjValue is still null, meaning it has not been set. However, if I decorate the property with the [Dependency] attribute, the container will inject it for you:

class MyPropertyDependencyClass

{

object objValue;

[Dependency]

public object ObjValue

{

get { return objValue; }

set { objValue = value; }

}

}

[Fact]

public void CanSetProperties()

{

DependencyContainer container = new DependencyContainer();

MyPropertyDependencyClass result = container.Get<MyPropertyDependencyClass>();

Assert.NotNull(result.ObjValue);

}

Now our test shows that the value is no longer null. The container has provided the dependency for us by setting the property.

### Method Injection

*Method Injection* works very much like constructor injection, except that the method is called after the object has been created (but before it’s handed back to the user, obviously).

By default, the container will not call any methods. You must decorate any methods you want to be called with [InjectionMethod].

class MyMethodDependencyClass

{

public object objValue;

[InjectionMethod]

public void Initialize(object obj)

{

objValue = obj;

}

}

[Fact]

public void CanInjectMethods()

{

DependencyContainer container = new DependencyContainer();

MyMethodDependencyClass result = container.Get<MyMethodDependencyClass>();

Assert.NotNull(result.objValue);

}

## Modifying Container Behavior

There are several options for modifying the way the container operates on your request.

### Type Mapping

By default, the container will attempt to create the exact type that was provided to the Get() method, as we’ve seen in all the previous examples. What happens if you want to take a dependency on an interface or an abstract type instead of a concrete type?

interface IFoo {}

[Fact]

public void CannotRequestInterface()

{

DependencyContainer container = new DependencyContainer();

IFoo result = container.Get<IFoo>();

}

If you run this test, it will fail with a message similar to this one:

failed: System.MissingMethodException : Cannot create an instance of an interface.

That seems like a reasonable complaint. It doesn’t know which implementation of the IFoo interface you really wanted, so it tells you it can’t fulfill your request.

You can override this behavior with *type mapping*:

interface IFoo {}

class Foo : IFoo {}

[Fact]

public void CanMapTypes()

{

DependencyContainer container = new DependencyContainer();

container.RegisterTypeMapping<IFoo, Foo>();

IFoo result = container.Get<IFoo>();

Assert.NotNull(result);

Assert.IsType<Foo>(result);

}

When you run this test, you can see that not only was it able to fulfill the request, we can verify that the thing we got back really was an instance of the Foo class.

### Singletons vs. Instances

By default, the container creates new instances of each object that’s requested, as illustrated by this test:

[Fact]

public void InstanceBehaviorIsDefault()

{

DependencyContainer container = new DependencyContainer();

object result1 = container.Get<object>();

object result2 = container.Get<object>();

Assert.NotSame(result1, result2);

}

You can achieve *singleton behavior* for any given request type:

[Fact]

public void ContainerCanUseSingletonBehavior()

{

DependencyContainer container = new DependencyContainer();

container.CacheInstancesOf<object>();

object result1 = container.Get<object>();

object result2 = container.Get<object>();

Assert.Same(result1, result2);

}

When specifying the singleton behavior, you can specify it for either the requested type or the mapped type. This allows you to say “any instance of IDatabase” is cached, or to say “only instances of FooDatabase are cached”.

Additionally, you can provide an instance of a singleton class for the container rather than wait for the container to create one on your behalf. For example:

[Fact]

public void CanProvideSingletonInstanceToContainer()

{

object obj = new object();

DependencyContainer container = new DependencyContainer();

container.RegisterSingletonInstance<object>(obj);

object result = container.Get<object>();

Assert.Same(obj, result);

}

## Event Broker

The container supports a loosely coupled eventing system called *Event Broker*, a concept first introduced in the [Composite UI Application Block](http://msdn2.microsoft.com/en-us/library/aa480450.aspx) (CAB). The implementation in DependencyContainer is a very simplified version of what was available in CAB.

There are two ways to use the event broker system: with code, and with attributes.

### Using Event Broker with Code

class MyEventSourceClass

{

public event EventHandler<EventArgs> MyEventSource;

public void Invoke()

{

if (MyEventSource != null)

MyEventSource(this, EventArgs.Empty);

}

}

class MyEventSinkClass

{

public bool eventSinkCalled;

public void MyEventHandler(object source, EventArgs e)

{

eventSinkCalled = true;

}

}

[Fact]

public void CanWireEventsWithCode()

{

DependencyContainer container = new DependencyContainer();

container.RegisterEventSource<MyEventSourceClass>("MyEventSource", "TheEventID");

container.RegisterEventSink<MyEventSinkClass>("MyEventHandler", "TheEventID");

MyEventSourceClass source = container.Get<MyEventSourceClass>();

MyEventSinkClass sink = container.Get<MyEventSinkClass>();

source.Invoke();

Assert.True(sink.eventSinkCalled);

}

The calls to RegisterEventSource() and RegisterEventSink() instruct the container that these classes are the sources and handlers of a named event, respectively. The event named MyEventSource on MyEventSourceClass, when raised, should be considered the event ID TheEventID. Similarly, the handler named MyEventHandler on MyEventSinkClass should be notified any time the event with ID TheEventID is raised.

When those two classes come into being together in the container, the event is wired to the handler automatically by the container.

The event broker implementation supports multiple sources and multiple sinks with the same ID. When multiple sources exist, then if any of them fires the event, the sink handlers are called. Similarly, if multiple sinks exist for the same event ID, they are all called when the event is fired. Note that when there are multiple sinks, the event broker system does not guarantee in which order they are called. It is important that the handlers not modify the event arguments, since the same instance of event arguments is shared among all the sinks.

### Using Event Broker with Attributes

You can use attributes to indicate event sources and sinks, like this:

class MyEventSourceClass

{

[EventSource("TheEventID")]

public event EventHandler<EventArgs> MyEventSource;

public void Invoke()

{

if (MyEventSource != null)

MyEventSource(this, EventArgs.Empty);

}

}

class MyEventSinkClass

{

public bool eventSinkCalled;

[EventSink("TheEventID")]

public void MyEventSink(object source,

EventArgs e)

{

eventSinkCalled = true;

}

}

[Fact]

public void CanWireEventsWithAttributes()

{

DependencyContainer container = new DependencyContainer();

MyEventSourceClass source = container.Get<MyEventSourceClass>();

MyEventSinkClass sink = container.Get<MyEventSinkClass>();

source.Invoke();

Assert.True(sink.eventSinkCalled);

}

This is the equivalent of our code example, except now we’re using attributes to decorate the event sources and sinks. The container finds these attributes and does the wiring on your behalf as the objects are created.

## Method Interception

TBD

## Miscellaneous

### Disposing the Container

Because the container owns the lifetime of some objects (namely, the things which are treated as singletons), it also exposes a Dispose method which will, in turn, dispose all the objects that it current owns the lifetime for.

### Nested Containers

The container supports nesting containers inside of one another. When

### Being Aware of Object Lifetime

On rare occasions, an object will need to know when it’s finished being injected. The container can alert an object when it’s completely finished being injected, although you should resist using this because it not only ties container knowledge into the object, it forces non-container users of the class to simulate the behavior of the container by calling the awareness methods.

Objects which need to be aware of their lifetime in the container can implement the IBuilderAware interface. The interface consists of two methods:

class MyBuilderAwareClass : IBuilderAware

{

public void OnBuiltUp(object buildKey)

{

Console.WriteLine("I was built using key " + buildKey);

}

public void OnTearingDown() {}

}

[Fact]

public void CanCreateBuilderAwareObjects()

{

DependencyContainer container = new DependencyContainer();

container.Get<MyBuilderAwareClass>();

}

If you run this test, you will see output which shows the requested build key (which for DependencyContainer is the originally requested build type).

For singleton objects, the objects are cleaned up when the container is disposed; for instance objects, the objects are cleaned up when you call TearDown on the container with the object instance. In either case, the OnTearingDown method is called before any of the tear down work is done.

### Configurators

TBD