Name: Olexandr Goloshchapov Date: 7/31/2020

Lab 5 – Dependency Injection

Objectives

Part 1 – Getting Started

Part 2 – Separating Object Instantiation

Part 3 – Supporting Mapping and Registration

Part 4 – Cleaning up the Object Registration Code

Part 5 – Supporting Singleton

Background/Scenario

In this lab, we will explore patterns and techniques for managing the complexity of object composition that can come from following the Dependency Inversion Principle. We will develop a generic Dependency Injection "container" (also known as an Inversion of Control container) so that we can realize the benefits of loosely coupled design in our own projects without the need to perform the manual "wiring" between our high- and low-level classes that would otherwise be required.

Overview

In our sample applications thus far, we have wired up our loosely coupled dependencies manually by creating and composing objects when the application starts. However, we also saw that constructor dependency injection requires more work up front to simply create an object and fulfill its dependency requirements.

While the manual approach is fine for a simple application that has a minimal number of objects, in a sufficiently large application, this object composition code can become complex. The number of parameters a constructor requires can rapidly grow as the class grows and comes to depend on more services. The order in which we construct objects and resolve dependencies must be carefully managed. We can quickly find ourselves having to deal with an unwieldy mess of boilerplate code just to get our application started.

As this construction activity becomes more painful, we may want to look to the functionality provided by a Dependency Injection Container to help manage the configuration of our dependencies and separate the responsibility of resolving those dependencies as we bootstrap our application. Containers can also help us deal with concerns related to lifetime management, such as knowing whether a request for an object should be resolved by instantiating a new object, by returning an existing instance of a singleton, or perhaps even by providing a separate object for each thread executing within the program.

More advanced containers can provide many other services, such as the ability to search for classes that can be used to fulfill dependencies and catalog them, both within the application's main assembly and even out in a directory on disk (think plug-in architecture). In this lab, we will focus just on the auto-wiring functionality that enables a container to perform dependency injection on the fly.

Required Resources

- Visual Studio 2017

Part 1: Getting Started

Download the DependencyInjection starter code from Canvas and familiarize yourself with the code base.

We are going to use a test-driven approach to help guide development of the methods and behavior of our container. Take a look at the provided MsTest project. Right now, these tests are commented out so that each can be taken step-by-step.

These tests will follow the naming pattern UnitOfWork_InitialCondition_ExpectedResult(). Uncomment out the test

```
GetInstance_ClassWithDefaultConstructor_ReturnsExpectedType()
```

to get started.

Part 2: Separating object instantiation

Step 1: GetInstance() basics

The first thing any container should be able to do is instantiate an object of a requested type. We will do this using reflection and the Activator.CreateInstance() method.

Add a method called GetInstance() that matches the expected interface of the test we enabled.

GetInstance() needs to return an instance of the type specified. Use the Activator.CreateInstance() method to do this. Our first test should now pass. We are well on our way to separating the concern of object instantiation from the rest of our code!

Go ahead and take this opportunity commit the progress you've made so far. Remember, you can never make too many commits.

Step 2: Making GetInstance() generic

It would be nice if we didn't have to use the typeof(ClassName) syntax to request instances of a type, or have to cast the return value from an object, so let's also provide a generic version of the GetInstance() method that simply calls our other GetInstance() method and results in a strongly-typed return value. Generics are very similar to templates in C++ and should be familiar.

Uncomment the

```
GetInstance_SupportsGeneric_ReturnsExpectedType()
```

unit test to get started. Once this tests passes, commit your progress.

Step 3: Supporting constructor parameters

So far, we've wrapped the construction of parameterless constructors, but that is only going to take us so far if we are trying to perform constructor dependency injection. Uncomment and run the unit test

```
GetInstance_ForConstructorThatNeedsParams_ReturnsExpectedType().
```

What happened?

Exception thrown: no parameterless constructor defined for this object. Looks like the "subject" is being assigned type of SomeClassA, but there is no constructor.

[Note: Use the Debug Unit Test option from the context menu (Ctrl+R, Ctrl+T) if you need to step through what the test is doing more carefully or simply run the test.]

Fortunately, Activator.CreateInstance() has an overload that looks like this:

```
public static object CreateInstance(Type type, params object[] args);
```

This is exactly what we need, but we're going to need to do some more reflection to build up to it.

Note in your GetInstance(Type type) method that type has a GetConstructors() method that returns an array of ConstructorInfo objects. For the purposes of our simple container, let's assume that we will always need to construct an object using the constructor that takes the most parameters.

- Write some code in your GetInstance() method that finds the Constructor that takes the most parameters. Again, step-debug into your method from the unit test if you need to inspect what the real data looks like for these classes at runtime. This will give a better idea of how to write the code you are going for.
- This is where things are about to get interesting, and a little recursive. Once we have a handle on the target constructor, we are going to need to ensure that we have instances of all of the objects that the constructor depends on. Fortunately, that's what our GetInstance() method does!
- Use your target ConstructorInfo's GetParameters() method to build up a list of arguments to pass to the Activator.CreateInstance() overload mentioned above. The type you need to construct is in the ParameterInfo's ParamaterType property. Strep through carefully if needed.

Once you provide instances of everything needed by the constructor, the unit test should pass.

When and how is SomeTypeA created in the

```
GetInstance_ForConstructorThatNeedsParams_ReturnsExpectedType()
```

unit test?

SomeClassA instance is created after the type and the number of parameters it takes is asserted through ParameterInfo and Type checking within GetInstance. Then, the values are passed to the overloaded CreateInstance function.

Now uncomment the GetInstance_ForClassWithNoArgsConstructor_ReturnsExpectedType unit test just to make sure we are still properly supporting classes that have a parameterless constructor, classes that have constructors with multiple arguments, and classes that use only the default constructor.

If all your tests pass, commit your progress.

Part 3: Supporting mapping and registration

Step 1: Type resolution

We've successfully separated the concern of instantiating objects out to our container, which is great. However, for our loosely coupled classes, when a constructor asks for an interface, we want to be able to resolve that into a concrete implementation.

For example, it would be great if I could ask my container for a Pet and get back a Dog, like so:

```
[TestMethod()]
public void GetInstance_WhenAskedForPet_ReturnsDog()
{
        //Arrange
        container.Register(typeof(IPet), typeof(Dog));
        //Act
        IPet pet = container.GetInstance<IPet>();
        //Assert
```

```
            pet.Should().BeOfType<Dog>();
            pet.MakeSound(); // Woof!
        }
```

This idea of “registering a type” that is associated with another type, and then providing an instance of one in place of the other, is powerful. It provides us a convenient way to wire up our loosely coupled classes without the need to manually create instances of everything, and it will clean up our object composition bootstrapping code significantly. To do this however, we are going to need to come up with a way to map a request for the in_type to a concrete instance of out_type.

So without further ado, uncomment the

```
GetInstance_WhenAskedForPet_ReturnsDog()
```

unit test, and let’s get started.

Step 2: Creating a type map dictionary

Add a Register() method to your Container class that matches the signature:

public void Register(Type in_type, Type out_type) { ... }

The next thing the Container class needs is a Dictionary. Dictionary can be found in the System.Collections.Generic namespace.

- Add a Dictionary as a private member that can be used to map from one type to another.
- Record the mapping in the dictionary when a client calls the Register() method.
- In GetInstance(), check if the type being requested is known about as an in_type in the registered type mappings. If it is, make the mapped out_type the target to be instantiated instead.
- Run all of the tests to make sure the earlier functionality is still working (Ctrl+R, Ctrl+A).

If all of that works, congratulations! You just built your first Inversion of Control/Dependency Injection container.

Step 3: Generic Register()

Let’s make it just a little bit nicer by implementing a generic method to wrap Register(), exactly as we did the GetInstance() method. GetInstanceGeneric_WhenAskedForPet_ReturnsCat() covers this, so go ahead and uncomment it now and write the generic wrapper.

Commit your progress.

Part 4: Cleaning up the object composition code

Our container is ready to use, so let’s take it for a spin in an example project. Locate the ArdalisRating example from class, navigate to UI\Program.cs : Main() and find the code that is manually wiring up all of the dependencies for RatingEngine. Note that we are already starting to see some complexity in the multiple places that we need to provide our ILogger Dependency. Let’s clean this up and rid ourselves of all these "new" statements with our new container:

- Add a Reference to the IoC library to ArdalisRating project.
- Add a new bootstrapper method called ComposeObjects() or RegisterTypes(). In practice, such a method might take a configuration for which Logger and Policy source to use (this is how we wire up different container profiles for different startup scenarios, similar to Factory pattern), but for now, just define it to be a no-parameter method that returns a new Container.
- Use the Register() method to create the following associations:

    - ILogger -> FileLogger
    - IPolicySource -> FilePolicySource
    - IPolicySerializer -> JsonPolicySerializer
- Call ComposeObjects() as a bootstrapping step at the start of Main() and capture the container it returns.
- Request a RatingEngine from the container using the GetInstance() method.

Step into your call to GetInstance() and follow along as it executes each step. What happened? Describe the sequence of events.

_____

_____

_____

_____

From here, we can go ahead and get rid of the code that explicitly creates new instances of FileLogger, RatingEngine, and RatingEngine's dependencies. Make sure the RatingEngine created by your container still runs with the rest of the application.

Commit your changes.

Part 5: Supporting Singleton

Notice that our container changed the previous singleton-like behavior of our Logger dependency for the two classes in ArdalisRating that depended on it. As we talked about in lecture, Singleton is often an anti-pattern that should be avoided in most cases that one might think to use it for (especially for globals), but there may still be situations where some classes should have only one instance and behave like a Singleton. With a few small but slightly more advanced changes, our container can support this very easily.

Step 1: RegisterSingleton()

The unit test

```
GetInstance_WhenAskedForSingleton_ReturnsSameInstance()
```

describes the behavior that we are going for, so uncomment it now. Add a generic method called RegisterSingleton() that is templated to take an instance of some type T as a parameter.

Now, we need some way to hold onto the object that we have been provided and return it when the type is requested. We could add a second lookup table for singleton types and search through it in our GetInstance() method before constructing a new type, but that would be pretty inefficient. Instead, let's refactor our existing mapping behavior to be a little more flexible.

Step 2: Using lambda expressions with the Dictionary

Lambda expressions provide a useful way to apply some additional instructions to our type lookup. (See Appendix A if you are new to lambda expressions.) We can use the Func type as the out value type in our dictionary lookup when we specify an in_type key. Then, we can use the wrapped method to apply a different strategy for resolving the object instance if the type was registered as a singleton rather than a regular type.

Modify the out_type parameter of your Dictionary<in_type, out_type> to instead be of type Func<object>. This is going to break some things, but keep going:

- Fix Register(): This probably no longer works because it is expecting an out_type for the value to go along with the in_type key. Instead, provide a lambda expression/anonymous method object here that wraps a call to GetInstance(out_type). What we are saying here is that, for types that are registered through the Register() method, we will still pass the out_type on to the normal GetInstance() creation sequence as before.

- Implement RegisterSingleton(): Now that we have the ability to call a function as part of our lookup, we can add the behavior we need to support singletons. When adding an entry to the Dictionary in RegisterSingleton(), specify the in_type typeof(T) as before, and for the out value, write a lambda expression that simply returns the instance that was provided to RegisterSingleton().

- Fix GetInstance(): Now the really fun part. In GetInstance(), if we have an entry in our dictionary for the type being requested, we should now call our anonymous method and return its result directly. For registered singletons, this will result in the original instance value that we wrapped in RegisterSingleton() with the lambda expression. For our regularly-registered types, the expression we provided in Register() has effectively already provided the lookup for out_type, and GetInstance() will be called again with it as the parameter.

If implemented properly, all of the unit tests should pass. Commit your progress.

Step 3: Wrapping up

We are almost there. The last step is to fix the Logger registration to behave as a singleton.

- Go back to the ComposeObjects() method and use the RegisterSingleton() method to configure the container to return a new instance of FileLogger that you provide it, as is done in the unit test.
- Run the RatingEngine application one last time to make sure everything still works.

If everything still works, commit your final changes, you're done with the code!

Reflection

1. When we updated the code to support singleton instances, we modified our dictionary from Dictionary<type, type> to Dictionary<type, Func<object>>. Why did we want to use Func<object> instead of just simply object (Dictionary<type, object>)? What would happen if we did use object instead?

   _____

   _____

   _____

   _____

2. Did you have any problems with this lab?

   _____

   _____

   _____

   _____

Appendix A – Lamba Expressions

In the Observer lab, we worked with delegate, which allowed us to wrap a function call in an object. C# provides another construct to wrap an anonymous method body into a delegate object in a similar fashion - essentially, you can define the body of a method inline, store it in a delegate, and pass it around like an object.

The language carries it even further and provides a much cleaner syntax for anonymous methods using lambda expressions, which allow one to define an anonymous method with a concise syntax that can be stored in an object and called at a later time. Here are some of the signatures of the object types that hold such an anonymous method:

▲ 1 of 17 ▼ Func<**out TResult**>
Encapsulates a method that has no parameters and returns a value of the type specified by the TResult parameter.
TResult: *The type of the return value of the method that this delegate encapsulates.*

▲ 3 of 17 ▼ Func<**in T1**, in T2, out TResult>
Encapsulates a method that has two parameters and returns a value of the type specified by the TResult parameter.
T1: *The type of the first parameter of the method that this delegate encapsulates.*

And so on…

The usage for an anonymous method that takes zero in parameters and returns one result looks like this:

```
Func<int> GetFortyTwo = () => { return 42; };
```

The right-hand part of => can be any expression that resolves to a value of type int. This expression can be reduced even further to something like:

```
Func<int> GetFortyTwo = () => 42;
```

Func<> behaves just like a strongly-typed function pointer and offers us similar access to deferred execution. With our method conveniently wrapped in an object, we can pass it around and call it like a function:

```
Console.WriteLine(GetFortyTwo());
```

Another powerful feature of lambda expressions is that they can enclose references to other variables that exist within the scope that they are declared, like so (see also Closures in C#):

```
int answer = 40;
Func<int> GetAnswer = () => answer + 2;
...
Console.WriteLine(GetAnswer());
```