Inversion Of Control

Estimated time for completion: 60 minutes

Overview:

In this lab you will refactor a class to make it testable. Once that is done you will write a dummy test class to inject into the SUT and then finally use an IoC container, configuring the container both in code and through the app.config file.

Goals:

- Refactor code to allow data access layer to be injected
- Use IoC and dependency injection to allow the code to be tested
- Understand how to use an IoC container
- Understand how to configure an IoC container

Lab Notes:

In this lab you will refactor a 'service' that currently has a hard coded dependency on data-access code. Dependencies such as this make code hard to test and difficult to extend. In the lab you will refactor the service and it's supporting classes to use a data layer abstraction and then use dependency injection to insert the data access code into the service

Part 1: Refactoring to allow a data access layer to be injected

In this part of the lab you will write code to test that the service interface is working. Initially the interface is hard coded to work against the database, you will change the code so that the data access mechanism can be injected into the service

1. The code has now expanded slightly, we've added a service layer that manages the database access for us. If you open the **QAndA.sln** file in the **InversionOfControl\before** directory you will see some extra projects. There is a **QAndADataAccessDAO**, and a **ConsoleQAndAService** project.

   The **ConsoleQAndAService** project provides a definition of a very simple service with just one method **VoteUpQuestion** to provide database access.

   If you look at the code in the console application you will see that it implements the interface and uses the (concrete) **QuestionDataAccessDAO** class to access the database. This is the code we want to test.

2. The console application is runnable but first you need to create the database. To create the database there is a file called **CSQAndADatabase.sql** in the **labs\sql** directory. Load this file into visual studio and create a connection to your local database instance (which may be SQLExpress). Once this is done execute the SQL.
3. If you are using SQLExpress you will need to change the connection strings in the TBlogServiceHost's app.config file. Look for the part of the string where it says "Data Source=.;" and change that to Data Source=./SQLExpress; There are two strings in the app.config, change both
4. At the moment the **Service** class in the console project is only testable if we are willing to use a real database and this is not acceptable if we want to run unit tests. Remember that unit test should be quick to run and use no external resources. There is a little work that needs doing to make this testable: we need to create an abstraction of the data access layer; provide a mock implementation of this layer; and create a test that uses this mock implementation.
5. The first step is to create an interface for the **QuestionDataAccessDAO** class, you want this interface to be in its own class library project, that way there is no dependency on the interface and whatever data access code you choose to implement. It's probably easer to do this in two steps.
6. Extract the interface.
    a. First use the Visual Studio refactoring to extract the interface from the **QuestionDataAccessDAO** class. Call the class **IQuestionDataAccess** and include all the methods (there is only one at the moment). Make the interface **public**
    b. Now create a new project, call it **QAndADataAccess** and move the interface definition across to that project making sure to change the **namespace** the interface is in.
    c. You will need to fix all the project references. Remember to add a reference to the **QAndADataAccess** project to the **ConsoleQAndAService** project
    d. Rebuild the code. Notice that we are flying blind at the moment because we have no unit tests, however, if you did create the database you could re-run the application and see if the result is the same, this is the equivalent of an integration test
7. Now that the interface is in place we need to change the console application to accept the interface. We are going to inject the interface and we have several choices; we could use setter injection, parameter injection or constructor injection. The best choice is nearly always constructor injection, there is generally less disruption to the code and it is possible to 'inject' a default value; that is what we will do here.
8. Add two constructors to the Service class, a default constructor and a constructor that takes an **IQuestionDataAccess** as a parameter. The default constructor should call the non-default constructor passing a new **QuestionDataAccessDAO** as the parameter (this is really a stopgap, eventually the default constructor can be removed).

```
private IQuestionDataAccess _questionDataAccess;
public Service()
    : this(new
   QuestionDataAccessDAO(ConfigurationManager.ConnectionStrings["QAndA"].Conn
   ectionString))
{}

public Service(IQuestionDataAccess questionDataAccess)
{
    _questionDataAccess = questionDataAccess;
}
```

9. Once that is in place remove all the calls to 'new QuestionDataAccessDAO()', use the _questionDataAccess member and execute the application again to ensure it still works.
10. We now need to define an interface for the service.
11. Create a class library project called **QAndAService** and add an interface called **IQAndAService**. This interface has one method called **VoteUpQuestion** that takes an int, the id of the question, i.e. it is the same method definition as exists on the **Service** class.
12. Once you've compiled the new project add a reference to this project to the **ConsoleQAndAService** project.
13. Make the **Service** class implement the **IQAndAService** interface.
14. We are now in a position to write a test for the service.
    Add a new project called ConsoleQAndAServiceTest.
    a. Create a new folder named 'lib' in this project
    b. In the top level tests directory there is a folder called lib, copy the nunit framwork DLL and the three Microsoft DLLs from this top level folder to the lib folder you just created.
    c. Add these DLLs as project references
15. Rename **Class1.cs** to **ConsoleTest.cs**
16. Make the class a **TestFixture** (make sure you use the correct namespace, **NUnit.Framework**) and add a **Test** method, call the method **ServiceShouldReturnCorrectNumberOfVotesFromVoteUpQuestion**
17. The first step in the test is to create an instance of the Service class, however to do this we need to have an instance of the **IQuestionDataAccess** interface. We cannot use the **QuestionDataAccessDAO** class as that would defeat the purpose, remember it does database access, so we have to create our own dummy implementation
    a. In the test project Create a new class called **DummyQuestionDataAccess** and have this class implement **IQuestionDataAccess.** You will need to make the class public and add the appropriate methods
    b. Give this dummy class a constructor that takes the initial number of votes to set on the question
    c. This dummy class needs to create a new Question that it can pass back from the call

```
public class DummyQuestionDataAccess : IQuestionDataAccess
{
    private readonly int _votes;
    public DummyQuestionDataAccess(int votes)
    {
        _votes = votes;
    }
    public Question GetQuestion(int id)
    {
        return new LowRatedQuestion(new User()) {Votes = _votes, Title =
  "Dummy title", Message = "Dummy Message"};
    }
}
```

18. You can now use this dummy in the test code
    a. In the test create a new **Service** instance and pass it a reference to the **DummyQuestionDataAccess** class
    b. Call the services **VoteUpQuestion** method
    c. **Assert** that the expected value is returned

```
[TestFixture]
public class ConsoleTest
{
    [Test]
    public void ServiceShouldReturnCorrectNumberOfVotesFromVoteUpQuestion()
    {
        IQAndAService service = new Service(new DummyQuestionDataAccess(1));
        int votes = service.VoteUpQuestion(1);

        Assert.That(2, Is.EqualTo(votes));
    }
}
```

19. You will need to add references to the appropriate libraries
20. This all works however there is an issue, what if you need more than one dummy class, creating multiple classes is awkward at best, we will address this limitation later

Part 2: Using an IoC container to initialize the service

In this part of the lab you will write code to test that the service interface is working. Initially the interface is hard coded to work against the database, you will change the code so that the data access mechanism can be injected into the service

1. Currently to use the service we have to hard code the data access class into the code, this is not very flexible

```
QuestionDataAccessDAO questionDataAccess = new
   QuestionDataAccessDAO(ConfigurationManager.ConnectionStrings["QAndA"].Conn
   ectionString);
```

2. In this part of the lab we will replace this by using the Microsoft Unity IoC container to create the instances. Initially we will do this in code and then using the configuration file.
3. Add a lib folder to the **ConsoleQAndAService** project, copy the Microsoft DLLs into this folder and add references to those DLLs to the project.
4. In the Service class we are going to initialize and use the Unity container, ideally you only want a single instance of the container and often the container is wrapped in a Singleton class to provide this behavior. In this case we will simple use a static container object
5. In the **Service** class a define a static type of **IUnityContainer** and initialize it to a new **UnityContainer**.

```
static IUnityContainer unityContainer = new UnityContainer();
```

6. Initially we are going to configure the unity container in code.
7. In the main method you need to register a mapping between the interface type and an instance of the interface. The instance you are going to register requires a constructor parameter so that has to be passed as well.
   In Main:
   a. Define a variable of type **InjectionConstructor** and create an instance passing it the connection string from the configuration file
   b. Now call the container's **RegisterType** method. This takes two generic parameters which are the interface and the implementation, and also the **InjectionConstructor** defined above

```
InjectionConstructor injectionConstructor = new InjectionConstructor(
           ConfigurationManager.ConnectionStrings["QAndA"].ConnectionString);
unityContainer.RegisterType<IQuestionDataAccess, QuestionDataAccessDAO>(
           new InjectionMember[]{injectionConstructor});
```

8. You can now use the container, you do this by replacing the default constructor where you currently create a new **QuestionDataAccessDAO**
   a. Instead of creating a new **QuestionDataAccessDAO** instance call the containers **ResolveType** method, this takes a generic parameter of the interface type to resolve.

```
public Service()
    : this(unityContainer.Resolve<IQuestionDataAccess>())
{ }
```

9. Run the application again, it should still work.

10. The next thing to do is to change the application so that it uses configuration to initialize the container. To do this you need to add configuration and then to write code to configure the container.
11. In the Service class add a static constructor to the code, in this constructor you need to initialize the container. The code looks like this

```
static IUnityContainer unityContainer;
static Service()
{
    unityContainer = new UnityContainer();
    UnityConfigurationSection section =
   ConfigurationManager.GetSection("unity") as UnityConfigurationSection;
    if (section != null)
    {
        section.Containers.Default.Configure(unityContainer);
    }

}
```

13. In Main remove the code that initializes the container
14. Once this code is in place you can add the configuration. In the labs\snippets directory there is an XML snippet, you can install this by copying the snippet to **[user]\ Documents\Visual Studio 2008\Code Snippets\XML\My Xml Snippets**.
15. Open **app.config**, near the top of the file, before the **connectionString** section, right click, select Insert Snippet and insert the unity snippet, you need to specify the names and locations of the interface and classes.
    You also need to add the constructor parameter to the configuration, the config file ends up looking like this:

```
<configuration>
  <configSections>
    <section name="unity"
   type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
   Microsoft.Practices.Unity.Configuration" />
  </configSections>
  <connectionStrings>
    <add name="QAndA"
   connectionString="server=.\SQLExpress;database=QAndA;trusted_connection=tr
   ue"/>
  </connectionStrings>
  <unity>
    <typeAliases>
      <typeAlias alias="IQuestionDataAccess"
   type="QAndADataAccess.IQuestionDataAccess, QAndADataAccess" />
      <typeAlias alias="QuestionDataAccessDAO"
   type="QAndADataAccessDAO.QuestionDataAccessDAO, QAndADataAccessDAO" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <type type="IQuestionDataAccess" mapTo="QuestionDataAccessDAO" >
            <typeConfig>
              <constructor>
                <param name="connectionString" parameterType="System.String">
                  <value
   value="server=.\SQLExpress;database=QAndA;trusted_connection=true"/>
                </param>
              </constructor>
            </typeConfig>
          </type>
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

16. The file maps **IQuestionDataAccess** to **QuestionDataAccessDAO**
17. Once that is done you can run the application again and it should still run

- Solution: after/qanda.sln