Design for Testing

Estimated time for completion: 60 minutes

Overview:

In this lab you will refactor a library of untestable code to make it testable (and more flexible). You will then use an IoC container to construct the application correctly via dependency injection

Goals:

- Understand the causes of tight coupling in applications
- Learn how to break dependencies with abstractions
- Learn how to wrap components outside of your control
- Understand how IoC containers help assemble the application

Lab Notes:

N/A

Part 1: Examine the starter solution

In the first part of the lab you will examine the starter solution and see how the tight coupling and use of statics make a library close to impossible to test.

1. Open **Before/DesignForTesting.sln**
2. The solution contains three projects:
3. **App** which is the application executable
4. **ThirdParty** which contains comonents which for this lab we are assuming cannot be changed
5. **Services** which contains the application logic and core components
6. If you open **CRM.cs** in the **Services** project you will see that there are a number of hard dependencies in in the code despite there being only one method
7. The use of `new` to construct the `ContactRepository`
8. The use of the `static` member of the `EmailUtil` class
9. The use of the `static` member of the `SMSUtil` class
10. In the `SMSUtil` and `EmailUtil` classes there are hard-wired dependencies on the third party components
11. Run the application and note the output. The goal is to refactor the code to make it testable and loosely coupled while not changing the functionality

Part 2: Decoupling the CRM and ContactRepository

In this part of the lab you will change the CRM code to use an abstraction for the contact respository and then inject it into the CRM constructor

1. Open **ContactRepository.cs**
2. Right click on the `ContactRepository` class name and select *Refactor/Extract Interface*
3. All of the methods are selected in the dialog so just press OK
4. Make sure the generated interface is marked as `public`
5. Go to **CRM.cs** and change the `repository` member variable to type `IContactRepository`. The `CRM` code is now, in effect, coded against an interface so all we need to do now is inject the dependency instead of using `new` and we will have broken our first tight coupling
6. Remove the creation of the `ContactRepository` (but not the member variable.
7. Initialize the `repository` member from a constructor parameter

```
IContactRepository repository;

public CRM(IContactRepository repository)
{
    this.repository = repository;
}
```

8. Compile the solution and you will find that the **App** project does not compile. Go to main and create an instance of the `ContactRepository`, then pass it into the `CRM` constructor
9. You should now be able to compile and run the application as before

Part 3: Decoupling the CRM and the Email and SMS Utils

In this part of the lab you will change the CRM code to use an abstraction for the contact respository and then inject it into the CRM constructor

1. Static members are an anathema to unit testing. As we control the code we will change the `Util` classes static methods into instance methods. Change the two `Util` classes so that the `Send` methods are instance methods (you will need to change the `EmailUtil` from being a static class as well to do this)
2. Look at the two `Util` classes – notice that the signature is very similar. The `EmailUtil SendEmail` method has an extra parameter but apart from that they are the same. If we could provide this extra parameter in another way then we could have a notion of a standard delivery service which means the `CRM` could be extended to take new forms of message delivery without changing the code. Make the `EmailUtil` class take the subject as a constructor argument and make the `Send` signatures the same

```
string subject;
public EmailUtil(string subject)
{
    this.subject = subject;
}
public void SendEmail(IEnumerable<Contact> contacts, string messageTemplate)
{
    foreach (Contact contact in contacts)
    {
        string message = string.Format(messageTemplate, contact.FirstName);

        gateway.Send("notspam@spam.net", contact.Email, subject, message);
    }
}
```

3. Extract an interface from `EmailUtil` with the `Send` method on it – call the interface `IDeliveryService`. Make suree the interface is marked as `public`
4. Make the `SMSUtil` class implement the `IDeliveryService` interface
5. The two `Util` classes are not really utility classes any more so rename them to rename them `EmailService` and `SMSService`.
6. Now we need to wire these into the `CRM` so add two constructor parameters each of `IDeliveryService`, one for the `SMSService` and one for the `EmailService` and store the these in fields

```
public class CRM
{
    IContactRepository repository;
    IDeliveryService smsService;
    IDeliveryService emailService;

    public CRM(IContactRepository repository, IDeliveryService smsService,
                                              IDeliveryService emailService)
    {
        this.repository = repository;
        this.smsService = smsService;
        this.emailService = emailService;
    }

    public void SendCustomerMessage(DeliveryMethod deliveryMethod,
                                    string messageTemplate)
    {
        switch (deliveryMethod)
        {
            case DeliveryMethod.SMS:
                smsService.Send(repository.GetAll(), messageTemplate);
                break;
            case DeliveryMethod.EMail:
                emailService.Send(repository.GetAll(), messageTemplate);
                break;
        }
    }
}
```

7. If you try to compile you will see that `App` still needs fixing up. Change `Main` to create instances of the `SMSService`, `EmailService` and `ContactRepository` and pass these into the `CRM` constructor.

```
IDeliveryService smsService = new SMSService();
IDeliveryService emailService = new EmailService("Great News");
IContactRepository repository = new ContactRepository();
var crm = new CRM(repository, smsService, emailService);
```

8. Compile and run the code and verify that the application runs as previously

---

Part 4: Wrapping up the Third Party Components

We still have hard wired dependencies in the SMSService and EmailService on the third party components. To allow us to ensure that these work correctly we need to be able to pass alternate versions of the gateways. This is what you will do in this part of the lab. Remember, however, that our constraint is that we cannot change the third party components as we don't control that code

1. If we cannot change a dependency then we have to wrap it. Introduce a new `interface` called `ISMSGateway` that has a `Send` method matching the third party `SMSGateway`
2. Add a class that implements `ISMSGateway` that simply hands off to the third party `SMSGateway`

```
public interface ISMSGateway
{
    void Send(string to, string message);
}

public class RealSMSGateway : ISMSGateway
{
    SMSGateway gateway = new SMSGateway();
    public void Send(string to, string message)
    {
        gateway.Send(to, message);
    }
}
```

3. Do the same for the `EmailGateway`

```
public interface IEmailGateway
{
    void Send(string from, string to, string subject, string message);
}

public class RealEmailGateway : IEmailGateway
{
    EmailGateway gateway = new EmailGateway();

    public void Send(string from, string to, string subject, string message)
    {
        gateway.Send(from, to, subject, message);
    }
}
```

4. Now inject the gateways into their appropriate `Services`. This will allow us to fake out the gateways during testing

```
private readonly ISMSGateway gateway;

public SMSService(ISMSGateway gateway)
{
    this.gateway = gateway;
}
```

5. Finally fix up `Main` to create instances of the real gateway wrappers and pass them into the services

```
ISMSGateway smsGateway = new RealSMSGateway();
IEmailGateway emailGateway = new RealEmailGateway();
IDeliveryService smsService = new SMSService(smsGateway);
IDeliveryService emailService = new EmailService("Great News", emailGateway);
```

6. Compile and run the code – it should function as before but is now all loosely coupled. However, there is a lot of construction logic in the application now – it would be good for something else to take responsibility for that. We will use an IoC container for this in the last part of the lab

---

Part 5: Using an IoC Container to Wire Up Dependencies

We want to remove the construction logic from the application to the point where we can declaraively say how abstractions map to concrete types and get the container to wire up the dependencies on our behalf. Sometimes this works very smoothly and sometimes you need to provide extra assistance as we shall see

1. Using **NuGet**, add the **Unity IoC container** to the **App** project
2. Add a helper method next to `Main` canlled `ConfigureContainer` that returns `IUnitContainer`. We will describe all of the dependencies in this method. For the time being just create an instance of `UnitContainer` and return it
3. In `Main` call `ConfigureContainer` and call `Resolve` for the `CRM` in place of using `new` to get the `CRM` instance

4. We will now configure the container. Lets start with some easy mappings: using `RegisterType`
    a. Map `ISMSGateway` to `RealSMSGateway`
    b. Map `IEmailGateway` to `RealEmailGateway`
    c. May `IContactRepository` to `ContactRepository`

```
container.RegisterType<ISMSGateway, RealSMSGateway>();
container.RegisterType<IEmailGateway, RealEmailGateway>();
container.RegisterType<IContactRepository, ContactRepository>();
```

5. Now, the `SMSService` and `EmailService` both implement the same interface so we will have to name those mappings to tell them apart. Register `IDeliveryService` to `SMSService` with a name of `SMS`

```
container.RegisterType<IDeliveryService, SMSService>("SMS");
```

6. The email service is a bit trickier as it takes a `string` as well as the email gateway so we will have to tell it how to resolve a constructor by passing an `InjectionConstructor` populated with the `string` and a `ResolveParameter` for `IEmailGateway` – the last tells the container to get the parameter by resolving the interface. Again use a mapping from `IDeliveryService` and name the mapping `Email`

```
container.RegisterType<IDeliveryService, EmailService>("Email",
          new InjectionConstructor("Great Offer",
                                   new ResolvedParameter<IEmailGateway>()));
```

7. Finally we need to wire up the `CRM` object dependencies. Here we need to state what gets wired up to each of the `IDeliveryService` parameters so again we need to be explicit about which dependencies go where. Use `RegisterType` to register `CRM` – it doesn't have an abstraction so you only need one generic parameter. Pass an `InjectionConstructor` that maps the correct parameters using `ResolveParameter` objects

```
container.RegisterType<CRM>(new InjectionConstructor(
                    new ResolvedParameter<IContactRepository>(),
                    new ResolvedParameter<IDeliveryService>("SMS"),
                    new ResolvedParameter<IDeliveryService>("Email")));
```

8. You should now be able to compile and run the application again. IoC wireup is often a lot simpler than this – just simple mappings – but sometimes you have to be more specific.
9. There are still oddities in some of the code at present that should probably be refactored. The delivery service should probably be passed in using parameter injection as needed for example – can you think of other ways to make the code more loosely coupled?

Solutions

after\DesignForTesting.sln

