# Dependency Injection: A Practical Introduction

*An overview of the dependency injection pattern by JeremyBytes.com*

## Overview

It's hard to turn around without hearing someone talking about Dependency Injection (at least if you are talking to other developers). But what exactly is Dependency Injection (DI)? And why would we want to use it?

It turns out that Dependency Injection is an extremely useful pattern in any non-trivial application (a small application will get limited benefit from DI). Some of the big benefits include extensibility, testability, and late binding.

We'll start our exploration of Dependency Injection by looking at a non-DI application. We'll see how conventional development can leave us with tightly-coupled code (even when we think we have good separation of concerns). Then we'll see how adding DI to the application adds the extensibility and testability to the application. Next, we'll take a look at two of the many DI containers that are available for us to use. Finally, we'll see how we can get late-binding by moving our container configuration from code to configuration (and the benefits and drawbacks of doing so).

We are just going to skim across the surface of Dependency Injection to get a good idea of some practical uses in our code. These examples are from my own coding experiences and show how Dependency Injection has been helpful for me. For a more in-depth look at DI, I highly recommend *Dependency Injection in .NET* by Mark Seemann.

Before we get started, I'll mention the S.O.L.I.D. principles. These are a set of 5 object-oriented design (OOD) principles talked about by Robert C. Martin (a.k.a. Uncle Bob). If you do some quick internet searches, you'll find all sorts of references to SOLID and OOD. We won't go into detail on the principles here, but I'll point out where they pop-up.

## What is Dependency Injection?

One of the issues with Dependency Injection is that there are dozens of definitions that describe the pattern just a little bit differently.

Wikipedia is a good place to start. And since it's nerds arguing with nerds, it's bound to have a pretty good definition. Here it is (at least at the time of this writing):

> *Dependency injection is a software design pattern that allows a choice of component to be made at run-time rather than compile time.*

Unfortunately, Wikipedia fails us a bit. It mentions run-time decisions (also called late binding), but Dependency Injection is much more than that.

So, let's see what Mark Seeman has to say:

> *Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code.*

I like this definition much better. Dependency Injection is all about how to create loosely coupled code (and this enables late binding, among other things).

So, we're looking at how we can create loosely-coupled code.

## Why Loosely-Coupled Code?

Loosely-coupled code helps us in a number of areas. Here are just a few:

### *Extensibility*

Extensibility is the ability to easily add new functionality to code. The "easily" part means that we can make updates in well-isolated areas rather than needing to update bits and pieces throughout the code base.

### *Late Binding*

As mentioned, late binding is the ability to choose what components we use at run-time rather than compile-time. We can only do this if our code is loosely-coupled – our code only cares about abstractions rather than a particular concrete type. This lets us swap components without needing to modify our code.

### *Parallel Development*

If our code is loosely-coupled, it makes it easier for multiple development teams to work on the same project. We can have one team working in the business layer, and a different team working in the service layer. Because the layers are independent, the teams will be in different source files that don't directly affect each other.

### *Maintainability*

When our components are independent, the functionality is isolated. This means that if we need to hunt down bugs or tweak functionality, we know exactly where to look.

### *Testability*

Unit testing is a hugely important topic. The primary goal of unit tests is to test small pieces of code in isolation. When we have loosely-coupled code, we can easily put in mock or fake dependencies so that we can easily isolate the parts of the code that we actually want to test.

Also, as mentioned earlier, we will run across several of the S.O.L.I.D. principles as we look at Dependency Injection.

## Dependency Injection Patterns

There are a number of design patterns that are used in DI.

- Constructor Injection
- Property Injection
- Method Injection
- Ambient Context
- Service Locator

We'll be looking at Constructor Injection and Property Injection here, since these are the primary patterns. You may want to look into the others as you get more comfortable with Dependency Injection.

This may sound a bit complicated, and you're probably wondering if you really want to get involved in Dependency Injection. In reality, these patterns and principles are not that complicated. We'll work our way into them slowly so that we have a good idea of what's going on.

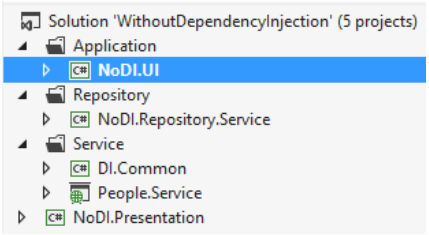
Some code samples will help us on our journey -- first, an application that does not use Dependency Injection.

# A Non-DI Sample

Our sample application will get data from a repository (using a WCF service). For the presentation layer, we'll use the MVVM (Model-View-ViewModel) pattern. (Don't worry if you aren't familiar with MVVM; we'll cover a few basics as we go.)

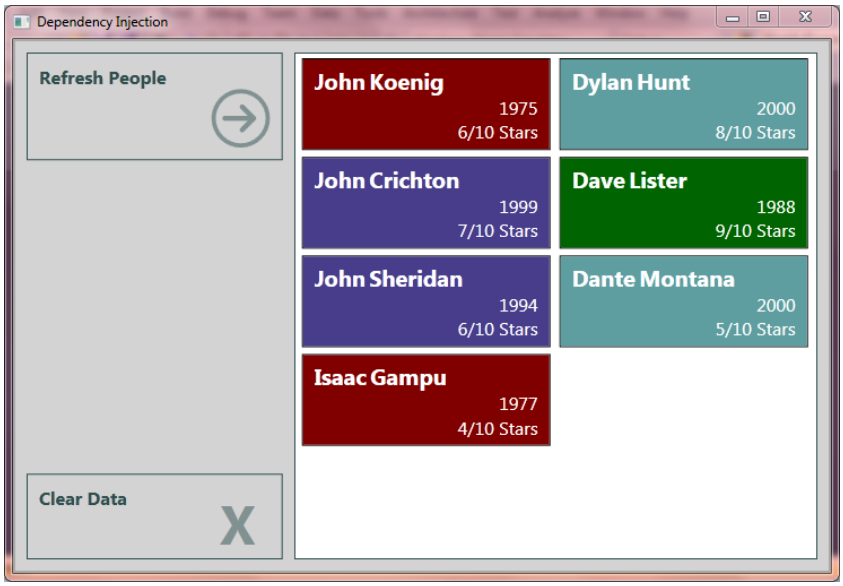
You can download the source code for the application from the website: http://www.jeremybytes.com/Downloads.aspx. The sample code we'll be looking at is built using .NET 4.5 and Visual Studio 2012 (however, everything will work with .NET 3.5 and .NET 4.0). The download consists of two solutions, each with multiple projects. Two versions are included: a "starter" solution (if you want to follow along) as well as the "completed" code.

To start with, we'll be using `WithoutDependencyInjection.sln`. Here's a quick overview of the projects:



- **NoDI.UI**
  A WPF application that contains our View (`MainWindow.xaml`).
- **NoDI.Repository.Service**
  A class library that contains the `PersonServiceRepository`. This is how data is retrieved from and persisted to the data store (through a WCF Service).
- **DI.Common**
  A class library that contains the definition of the `Person` class. (It also contains some interfaces that are not used in this solution.)
- **People.Service**
  A WCF service that acts as our persistence layer for the data.
- **NoDI.Presentation**
  A class library that contains our ViewModel (`MainWindowViewModel.cs`).

We'll look at each of these projects more closely as we go. First, let's run the application. When we click the "Refresh People" button, we get the following:




Let's start at the service layer and work our way to the UI.

### *The Service Layer - People.Service*

In our `People.Service` project, we have `PersonService.svc.cs` (details abbreviated):

```
public class PersonService : IPersonService
{
    public List<Person> GetPeople()...

    public Person GetPerson(string lastName)...

    public void AddPerson(Person newPerson)...

    public void UpdatePerson(string lastName, Person updatedPerson)...

    public void DeletePerson(string lastName)...

    public void UpdatePeople(List<Person> updatedPeople)...
}
```

For simplicity, this service returns hard-coded values for the `GetPeople` method (the method we'll be using in these samples). The Person class simply has four properties (from `Person.cs` in `DI.Common`):

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime StartDate { get; set; }
    public int Rating { get; set; }
}
```

### *The Repository Layer - NoDI.Repository.Service*

Next, we'll look at the `NoDI.Repository.Service` project. First, note that this project contains a Service Reference to the `PersonService` above. The `PersonServiceRepository` is an implementation of the repository pattern. The idea behind the repository pattern is that we can put a layer of abstraction between our application and the data storage layer – this will become clearer when we get to our DI examples a bit later.

For now, the `PersonServiceRepository` contains methods that look very similar to the service methods:

```
public class PersonServiceRepository
{
    public IPersonService ServiceProxy { get; set; }

    public PersonServiceRepository()
    {
        ServiceProxy = new PersonServiceClient();
    }

    public IEnumerable<Person> GetPeople()
    {
        return ServiceProxy.GetPeople();
    }
```

```
        public Person GetPerson(string lastName)
        {
            return ServiceProxy.GetPerson(lastName);
        }

        public void AddPerson(Person newPerson)...

        public void UpdatePerson(string lastName, Person updatedPerson)...

        public void DeletePerson(string lastName)...

        public void UpdatePeople(IEnumerable<Person> updatedPeople)...
    }
```

Notice that in the constructor, we are "new"ing up an instance of `PersonServiceClient` (our WCF service proxy).

### *The View Model Layer - NoDI.Presentation*

`NoDI.Presentation` contains our View Model. When using the MVVM pattern, the "VM" (ViewModel) is the part of the presentation layer that is responsible for exposing properties and commands that the UI can use for data binding. The View Model gets data from the Model (in this case, our repository) and then exposes a set of properties that can be used by the View (which we'll see in just a bit).

Here is our (slightly abbreviated) code from `MainWindowViewModel.cs`:

```
    public class MainWindowViewModel : INotifyPropertyChanged
    {
        protected PersonServiceRepository Repository;

        private IEnumerable<Person> _people;
        public IEnumerable<Person> People
        {
            get { return _people; }
            set
            {
                if (_people == value)
                    return;
                _people = value;
                RaisePropertyChanged("People");
            }
        }

        public MainWindowViewModel()
        {
            Repository = new PersonServiceRepository();
        }
```

```
        //RefreshCommand Standard Stuff...

            public void Execute(object parameter)
            {
                ViewModel.People = ViewModel.Repository.GetPeople();
            }

        // ClearCommand Standard Stuff...

            public void Execute(object parameter)
            {
                ViewModel.People = new List<Person>();
            }

        // INotifyPropertyChanged Members...
    }
```

A few things to note here: first, notice that our class implements `INotifyPropertyChanged`. This is an interface that is used for data binding in WinForms and XAML (WPF, Silverlight, Windows Phone, etc.); it ensures that the UI is properly notified when the underlying data values are changed. The implementation for `INotifyPropertyChanged` is pretty boiler-plate, so it has been excluded from this snippet.

Next, we have a class-level variable for the `PersonServiceRepository` (note that the constructor instantiates this variable).

We also have a `People` property. This contains our actual collection of `Person` objects that are returned from the repository. These will be used to populate the list box in the UI.

Finally, we have two commands. I've only included the interesting bits here – the `Execute` method for each of our commands. `RefreshCommand`'s `Execute` method calls the `GetPeople` from the repository and uses the results to populate the `People` property. In contrast, the `ClearCommand` resets the `People` property to an empty collection.

Commands are used here because they can be easily data bound to buttons in XAML. This lets us minimize the code-behind that we have in our View files (which we'll see in just a moment). If you want more details, you can look up "Commanding Overview" in Visual Studio Help or MSDN.

### *The View Layer - NoDI.UI*

`NoDI.UI` is our WPF application. This contains the View part of our MVVM implementation. As mentioned earlier, MVVM works by allowing the View (our XAML) to data bind to properties exposed in our View Model. This keeps the code in our View to a minimum and gives us good separation between our presentation (the XAML) and the logic that drives it (the view model). For more information on MVVM, you can take a look at "Overview of the MVVM Design Pattern" on my blog: http://jeremybytes.blogspot.com/2012/04/overview-of-mvvm-design-pattern.html.

If we look at `MainWindow.xaml`, we can see where items are data bound to our view model:

```
<!-- Refresh List Button -->
<Button x:Name="RefreshButton" Grid.Column="0" Grid.Row="0" Margin="5"
                Content="Refresh People"
                Command="{Binding RefreshPeopleCommand}"
                Style="{StaticResource GoButtonStyle}" />
<!-- Clear Button -->
<Button x:Name="ClearButton"
                Grid.Column="0" Grid.Row="4"
                FontSize="16" Padding="7,3" Margin="5"
                Content="Clear Data"
                Style="{StaticResource ClearButtonStyle}"
                Command="{Binding ClearPeopleCommand}" />
<!-- List Box -->
<ListBox x:Name="PersonListBox"
         Grid.Column="1" Grid.Row="0" Grid.RowSpan="5"
         Margin="5"
         BorderBrush="DarkSlateGray" BorderThickness="1"
         ScrollViewer.HorizontalScrollBarVisibility="Disabled"
         ItemsSource="{Binding People}"
         ItemTemplate="{StaticResource PersonListTemplate}">
    <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapPanel />
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
</ListBox>
```

Notice that the `Command` properties for the buttons are data bound to the appropriate commands in the view model. With this binding in place, when we click the "Refresh People" button, the `Execute` method of the `RefreshPeopleCommand` will fire. For the `ListBox`, the `ItemSource` property is data bound to our `People` property from the view model.

The last step is to associate the View (`MainWindow.xaml`) with the View Model (`MainWindowViewModel.cs`). This is done in the code-behind – `MainWindow.xaml.cs`:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        ViewModel = new MainWindowViewModel();
    }

    public MainWindowViewModel ViewModel
    {
        get { return (MainWindowViewModel)this.DataContext; }
        set
        {
            this.DataContext = value;
        }
    }
}
```

Here, we have a property (`ViewModel`) which is of type `MainWindowViewModel`. We set this property to the `DataContext` of the window (`this`). The constructor creates a new instance of `MainWindowViewModel` and sets it to the `ViewModel` property. This has the effect of setting the data context (i.e. our data binding source) of our entire window to the `MainWindowViewModel`. This is how our Binding statements in the XAML know where to find the properties that they bind to.

## This Looks Pretty Good, Doesn't It?

Okay, so that was quite a bit of code to go through. Here's a review of the layers:

- View Layer
  The UI elements (XAML)
- View Model Layer
  The presentation logic
- Repository Layer
  The data interaction logic
- Service Layer
  The actual data access

And at first glance, it looks like we're adhering to some good object-oriented design principles – such as the Single Responsibility Principle (the S in S.O.L.I.D.). The Single Responsibility Principle states that an object should have one (and only one) reason to change – meaning that it does one thing and does it well.

In our project, we have a separation of concerns – the view, view model, repository, and service are all in their own separate classes. This helps with maintainability because we know where to make updates if we need to make changes. If it is related to the presentation logic, it goes in the view model; if it is related to the data interaction, it goes in the repository.

## The Illusion

Having this separation of concerns would lead us to believe that this code is loosely-coupled. After all, everything is separated out and has its own place, right? Unfortunately, this is not the case. In reality, our code is very tightly coupled. Let's walk the code (this time from the View end) to find our problem.

### *The View is Tightly-Coupled to the View Model*

Consider this piece of code from our View:

```
ViewModel = new MainWindowViewModel();
```

Because the constructor of the View is "new"ing up a specific instance of `MainWindowViewModel`, it is tightly-coupled to that class. It has a direct reference to the View Model, and we cannot build our View unless the assembly for `MainWindowViewModel` is also present at compile time. But things get worse.

### *The View Model is Tightly-Coupled to the Repository*

Consider this piece of code from our View Model:

```
Repository = new PersonServiceRepository();
```

Because the constructor of the View Model is "new"ing up a specific instance of `PersonServiceRepository`, it is tightly-coupled to that class. But it gets even worse.

### *The Repository is Tightly-Coupled to the Service*

Consider this piece of code from our Repository:

```
ServiceProxy = new PersonServiceClient();
```

Because the constructor of the Repository is "new"ing up a specific instance of `PersonServiceClient`, it is tightly-coupled to that class. And this leads us to a horrible realization.

### *The View is Tightly-Coupled to the Service*

Because of all of the tight-coupling, the View is tightly-coupled to the Service (through the View Model and Repository). YIKES!

Granted, the application that we have here is trivial, and this tight-coupling might not seem that bad. After all, with an application of this size, it isn't that difficult to modify and keep track of everything that is going on. But we're just using this as a simple example that we can easily wrap our heads around without getting too caught up in complex business logic. When we start to look at applications of significant size, then things change immensely.

## The Result of Tight-Coupling

So why should we care about this tight coupling? Here are a few scenarios to consider (which we will address with Dependency Injection).

### *Scenario 1: An Additional Repository*

Application requirements are ever-changing. With experience, we get to know the areas that are more likely to change (depending on our business environment). For this example, I can easily imagine that we will want to add other data storage options for our clients – perhaps by saving to a SQL Server (or other database) or to a text file (either CSV or XML).

But how would we handle an additional repository with this code? We would need to modify the View Model so that it would instantiate a `PersonServiceRepository` or a `PersonSQLRepository` or a `PersonCSVRepository`. And each time we add a new repository type, we would need to modify our View Model again. Our goal should be to eliminate the tight-coupling so that we can add a Repository without needing to modify (or even recompile) our View Model.

### *Scenario 2: A Caching Repository*

Calls to the data store are "expensive" (traveling across the wire and talking to another server), especially on a mobile device. Wouldn't it be great if we could cache some of the data on the client

side? That way, we would not have to make network calls each time we need to access data that seldom changes.

We can easily build a caching repository that will handle this for us, but we run into the same problem as above – we need to modify our View Model code to accept this new repository. Plus, we need some way to make our caching repository flexible so that it will work with the service repository or the SQL repository or the CSV repository.

### *Scenario 3: Unit Testing*

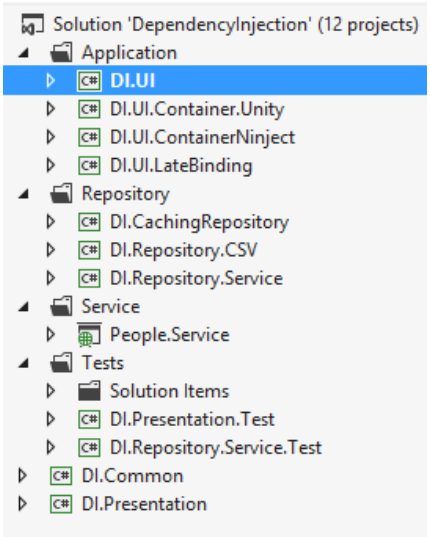
One big thing that is missing from our current solution is Unit Testing. Unit Testing allows us to automate testing of isolated sections of our code. But since our current code is tightly-coupled, we have no easy way to isolate those sections for testing. (It's not impossible, but it would be much easier to refactor the code than to put together the test harness required to isolate functions). We need to be able to test our View Model without the Repository code getting in the way. And we need to be able to test our Repository code without the Service code getting in the way.

So, in short, our goal should be to add some "seams" that allow us to isolate the code for testing without bringing in all of the tightly-coupled dependencies that we have now.

## Dependency Injection to the Rescue

Dependency Injection can help us address these scenarios to make our code easily extensible, improve testability, and even include runtime changes that don't require us to recompile the application.

Let's take a look at the other solution in our sample code: `DependencyInjection.sln`:




These projects are similar to the projects from the previous solution, so we won't go through complete descriptions here. The basic elements are the same – with some variations that we'll see as we go along.

Let's start by using Dependency Injection to break up some of our tight-coupling.

## Injecting the Repository into the View Model

One of the principles of good object-oriented design is to program to an interface rather than a concrete type. We can break the tight-coupling between the View Model and the Repository by adding a repository interface. This will add a layer of abstraction between our two components. For more information on Interfaces, please see "IEnumerable, ISaveable, IDontGetIt: Understanding .NET Interfaces" (available here: http://www.jeremybytes.com/Demos.aspx#INT).

The `DI.Common` project contains the `IPersonRepository` interface. This interface was extracted from `PersonServiceRepository`. With this abstraction, we can implement any number of concrete repositories.

```
public interface IPersonRepository
{
    IEnumerable<Person> GetPeople();

    Person GetPerson(string lastName);

    void AddPerson(Person newPerson);

    void UpdatePerson(string lastName, Person updatedPerson);

    void DeletePerson(string lastName);

    void UpdatePeople(IEnumerable<Person> updatedPeople);
}
```

Our first concrete repository is the one we already have: `PersonServiceRepository`.

```
public class PersonServiceRepository : IPersonRepository
```

Now that we have an interface, we can create additional repository classes that implement the same interface. The `PersonCSVRepository` is just such a class. You can check the class in the `DI.Repository.CSV` project for the implementation details.

> *A side note about the Repository interface: IPersonRepository will only work for the Person class. If we had another object type (such as Product), we would need to create a separate interface. As an alternative, we could use an interface with generic parameters to create an interface that can be used across types. IRepository<T, TKey> is just such an interface and has been included for reference. For more information on Generics, see "T, Earl Grey, Hot: Generics in .NET" (at http://www.jeremybytes.com/Demos.aspx). For this example, we are using IPersonRepository for simplicity.*

### *Constructor Injection*

Now that we have our abstraction, the only thing left is to add it as an injected dependency to our View Model. As a reminder, here's what our `MainWindowViewModel.cs` looked like before:

```
public class MainWindowViewModel : INotifyPropertyChanged
{
    protected PersonServiceRepository Repository;

    public MainWindowViewModel()
    {
        Repository = new PersonServiceRepository();
    }
    ...other members removed
}
```

Rather than having the constructor "new" up a concrete type, we'll pass an instance in to the class as a constructor parameter:

```
public class MainWindowViewModel : INotifyPropertyChanged
{
    protected IPersonRepository Repository;

    public MainWindowViewModel(IPersonRepository repository)
    {
        Repository = repository;
    }
    ...other members removed
}
```

First, notice that we have changed our `Repository` variable type from `PersonServiceRepository` to `IPersonRepository` – we are using an abstraction (the interface) rather than a concrete type. This allows the `Repository` variable to accept any class that implements the `IPersonService` interface (including `PersonServiceRepository`, `PersonCSVRepository`, plus any other new repositories we may create in the future). This makes our code extensible while remaining unchanged. This is referred to as the Open-Closed Principle (the O in the S.O.L.I.D. principles) – the code is open for extension but closed for modification.

These changes also help us adhere to the Dependency Inversion Principle (the D in S.O.L.I.D.) – abstractions should not depend upon details; details should depend upon abstractions. In our code, instead of depending on a concrete type, we have shifted our dependency to an abstraction (our `IPersonRepository` interface).

But how do we get the concrete type that implements the interface? We make it "somebody else's problem". We've already determined that the View Model doesn't need to know anything about the implementation details of the repository. All it cares about is that the repository implements `IPersonRepository`. So, if the View Model should not be responsible for the specific repository, who should be? We'll find this out in a bit.

To get the repository into our class, we have added a constructor parameter that uses our interface. This method is known as *Constructor Injection*. This is a preferred method for injecting dependencies into a class. The primary advantage is that we know exactly what dependencies need to be fulfilled simply by looking at the constructor.

### *Removing Dependencies*

Since we have removed the dependency on a concrete repository (`PersonServiceRepository`), we can also remove the references to that assembly.

First, we can remove the using statement:

```
using DI.Repository.Service;
```

Next, we can remove the assembly reference. Just find the `DI.Repository.Service` assembly in the References, right-click it, and select "Remove". Now our View Model is completely isolated from any concrete repository. It only cares about the abstraction – the `IPersonRepository` interface.

Now, that's loose coupling!

## Injecting the View Model into the View

So far, so good. We've managed to de-couple our View Model from a specific implementation of the Repository. But now we have another problem. Our code won't build. Since we removed the default (no parameter) constructor from the View Model, our View won't compile. As a reminder, here is our `MainWindow.xaml.cs` code (from the `DI.UI` project):

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        ViewModel = new MainWindowViewModel();
    }

    public MainWindowViewModel ViewModel
    {
        get { return (MainWindowViewModel)this.DataContext; }
        set
        {
            this.DataContext = value;
        }
    }
}
```

We get an error on the following line:

```
            ViewModel = new MainWindowViewModel();
```

Now, we could add the required parameter to the `MainWindowViewModel` constructor. This would mean creating an instance of the Repository we want to use, but this really wouldn't accomplish our goals. We've already said that the View Model should not be responsible for the repository. If the View Model should not be responsible, then the View definitely should not be responsible for it.

Instead, we'll use Constructor Injection on the View as well (to inject the View Model). This will make it "somebody else's problem". Here's our updated View code:

```
public partial class MainWindow : Window
{
    public MainWindow(MainWindowViewModel viewModel)
    {
        InitializeComponent();
        ViewModel = viewModel;
    }

    public MainWindowViewModel ViewModel
    {
        get { return (MainWindowViewModel)this.DataContext; }
        set
        {
            this.DataContext = value;
        }
    }
}
```

This is great; we're now injecting the View Model dependency into our View, so the View is no longer responsible for creating the instance. We could take this a step further and extract an interface from our View Model, but we won't do that here. In a larger system where View and View Models tend to be more mix-and-match, this would be an excellent idea. But we'll just stick with the concrete type in our sample.

## Composing the Dependency Graph

So, this hasn't really solved our problem, either. When we build, our application builds successfully. But if we run, we immediately get a run-time error. And the error is non-helpful: "Object not set to instance of an object."

When a WPF application starts, it looks for the `StartupUri` (in the `App.xaml` file). This points to the XAML that will be used as the application's main window. The issue is that WPF expects that this startup object has a default (no parameter) constructor. Since we removed our default constructor from `MainWindow.xaml.cs`, we need to create the window a bit differently. Here's what we'll do – in the `App.xaml` file, remove the following attribute from the opening `Application` tag:

```
StartupUri="MainWindow.xaml"
```

Now, we'll add a bit of code to `App.xaml.cs`:

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        Application.Current.MainWindow = new MainWindow();
        Application.Current.MainWindow.Show();
    }
}
```

This is the equivalent of using the StartupUri. The OnStartup event fires as the application is starting. What we need to do is instantiate the MainWindow and then show it. And that's what we're doing.

In the second line of the method, we are "new"ing up our `MainWindow` class (from `MainWindow.xaml`) and then assigning it to the `MainWindow` property of our `Application`. Once the `MainWindow` property is set, we can `Show` it. This displays that form, and since it is the `MainWindow`, when that form is closed, the application will shut down.

One thing about this code snippet: even though it is an equivalent replacement for the `StartupUri` that we had before, it still won't build. That's because it is still trying to call the `MainWindow` constructor without any parameters. So, let's fix that.

Our `MainWindow` constructor needs a `MainWindowViewModel` as a parameter. Let's create one of those and pass it to the constructor:

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
    var viewModel = new MainWindowViewModel();
    Application.Current.MainWindow = new MainWindow(viewModel);
    Application.Current.MainWindow.Show();
}
```

This is bit better, but it still won't build. Our `MainWindowViewModel` constructor needs an `IPersonRepository` parameter. Let's add that:

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
    var repository = new PersonServiceRepository();
    var viewModel = new MainWindowViewModel(repository);
    Application.Current.MainWindow = new MainWindow(viewModel);
    Application.Current.MainWindow.Show();
}
```

Now our application will build successfully. But let's refactor this just a bit before we go. The code that we just added is where we are composing our objects – we're snapping our loosely-coupled pieces together. So, we'll put those into a method to make that more apparent:

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
    ComposeObjects();
    Application.Current.MainWindow.Show();
}

private static void ComposeObjects()
{
    var repository = new PersonServiceRepository();
    var viewModel = new MainWindowViewModel(repository);
    Application.Current.MainWindow = new MainWindow(viewModel);
}
```

This gives us what is referred to as a *Composition Root*. This is the location where we will wire up all of the dependencies for our object graph.

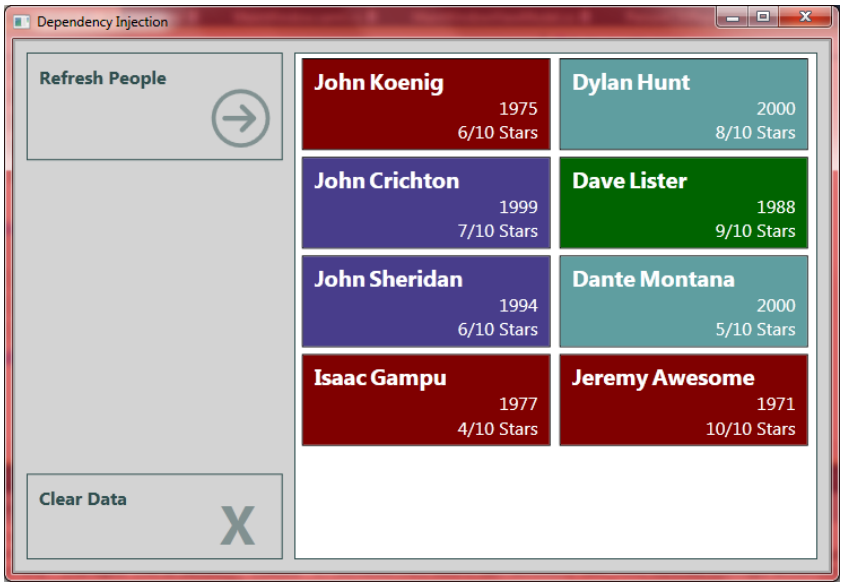
If we run the application, we will see the same results that we had earlier. But here's the difference: we've removed the tight-coupling from our code. The View Model is no longer tightly-coupled to the `PersonServiceRepository`. And because we broke this coupling, our View is no longer tightly-coupled to the Service.

In fact, we can easily swap out the repository by changing one line of code in our `ComposeObject` method:

```
var repository = new PersonCSVRepository();
```

Our old code would have required that we make changes to the View Model in order to use a different Repository. But since we've broken the coupling (and used DI to inject the Repository), the View Model no longer cares about the concrete type it is using as long as that type implements the appropriate interface (`IPersonRepository`). So, no matter how many new repositories we add, we will never have to update our View Model (whoo hoo!).

If we make these changes for the CSV repository and run the application, we get a slightly different result:




There is an additional Person (Jeremy Awesome) in the CSV file. This is so that it is easy to tell if we are using the Service repository or the CSV repository.

Let's go ahead and swap back to the `PersonServiceRepository` before continuing to the next example.

## Caching Repository

Above, we talked about adding a caching repository – this would keep a client-side copy of the data so that we don't have to make a server call each time we ask for data. Another requirement is that the caching repository needs to work with whatever concrete repository we want – whether the `PersonServiceRepository`, `PersonCSVRepository`, or some other repository. This makes it an ideal candidate for using a Decorator Pattern with Dependency Injection.

### *The Decorator Pattern*

The Decorator Pattern is a very simple idea. It describes a way for us to wrap an existing type, add some functionality, and expose the same interface as the original type. This makes it a drop-in replacement for the existing type.

> *As an aside, this could be seen as adhering to the Liskov Substitution Principle (the L in S.O.L.I.D.) – subtypes must be substitutable for their base types. Technically, we are using object composition and interfaces rather than direct inheritance, but the 100%-compatible object replacement fits the general principle.*

Let's see how this works. Our solution contains a `CachingServiceRepository` class (in `DI.Caching.Repository`):

```
public class CachingPersonRepository : IPersonRepository
{
    private TimeSpan _cacheDuration = new TimeSpan(0, 0, 30);
    private DateTime _dataDateTime;
    private IPersonRepository _personRepository;
    private IEnumerable<Person> _cachedItems;

    public CachingPersonRepository(IPersonRepository personRepository)
    {
        _personRepository = personRepository;
    }

    private bool IsCacheValid
    {
        get
        {
            var _cacheAge = DateTime.Now - _dataDateTime;
            return _cacheAge < _cacheDuration;
        }
    }

    private void ValidateCache()
    {
        if (_cachedItems == null || !IsCacheValid)
        {
            try
            {
                _cachedItems = _personRepository.GetPeople();
                _dataDateTime = DateTime.Now;
            }
```

```
            catch
            {
                _cachedItems = new List<Person>()
                {
                    new Person(){ FirstName="No Data Available",
                                  LastName = string.Empty, Rating = 0,
                                  StartDate = DateTime.Today},
                };
            }
        }
    }

    private void InvalidateCache()
    {
        _dataDateTime = DateTime.MinValue;
    }

    ... interface implementation skipped (for now)
}
```

First, notice that `CachingPersonRepository` implements `IPersonRepository`. This is important so that we can use it as a drop-in replacement where we would use any of our other repositories. Next, we have a series of variables to help us keep track of the cache.

- `_cacheDuration` – Specifies how long the cache should be valid. In our case, this is set to 30 seconds (so that we can see the change relatively quickly). In a real-world app, we would probably want to pass this in as a parameter.
- `_dataDateTime` – Specifies the date/time that our data was last updated. This is used to calculate whether the cache is still valid.
- `_personRepository` – This is our "wrapped" repository. If we look at the constructor, we can see that this instance is injected.
- `_cachedItems` – This is our client-side data cache.

Next, we have a constructor with our real repository instance injected. Then, we have the `ValidateCache` method with our caching logic. This does some quick math to determine whether the current `_cachedItems` is still valid. If it is not valid, then we make a call to the wrapped repository's `GetPeople` method to refresh the cache. Notice the `catch` block: if we can't successfully call the wrapped repository (for example, loss of network connectivity), then a default "No Data Available" record is returned.

Next, we have an `InvalidateCache` method. This allows us to force an update of the data on the next access.

To see how these methods are used, let's look at the `IPersonRepository` implementation:

```
    public IEnumerable<Person> GetPeople()
    {
        ValidateCache();
        return _cachedItems;
    }
```

```
public Person GetPerson(string lastName)
{
    ValidateCache();
    return _cachedItems.FirstOrDefault(p => p.LastName == lastName);
}

public void AddPerson(Person newPerson)
{
    _personRepository.AddPerson(newPerson);
    InvalidateCache();
}

public void UpdatePerson(string lastName, Person updatedPerson)
{
    _personRepository.UpdatePerson(lastName, updatedPerson);
    InvalidateCache();
}

public void DeletePerson(string lastName)
{
    _personRepository.DeletePerson(lastName);
    InvalidateCache();
}

public void UpdatePeople(IEnumerable<Person> updatedPeople)
{
    _personRepository.UpdatePeople(updatedPeople);
    InvalidateCache();
}
```

For our two "Get" methods, we call `ValidateCache` (which will refresh the client-side data, if required) and then return data based on the client-side data.

For the methods that modify data, we call the wrapped repository's methods and then call `InvalidateCache`. This will force us to get fresh (updated) data from the wrapped repository on the next "Get" call.

So overall, this code is not very complex. But because we are injecting the underlying repository (through Constructor Injection), this will work with any concrete repository we wish.

### *Composing Objects with the Caching Repository*

In order to use the caching repository, we just need to change our Composition Root (in `App.xaml.cs`). Here's our updated code:

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        ComposeObjects();
        Application.Current.MainWindow.Show();
    }
```

```
        private void ComposeObjects()
        {
            var wrappedRepository = new PersonServiceRepository();
            var repository = new CachingPersonRepository(wrappedRepository);
            var viewModel = new MainWindowViewModel(repository);
            Application.Current.MainWindow = new MainWindow(viewModel);
        }
    }
```

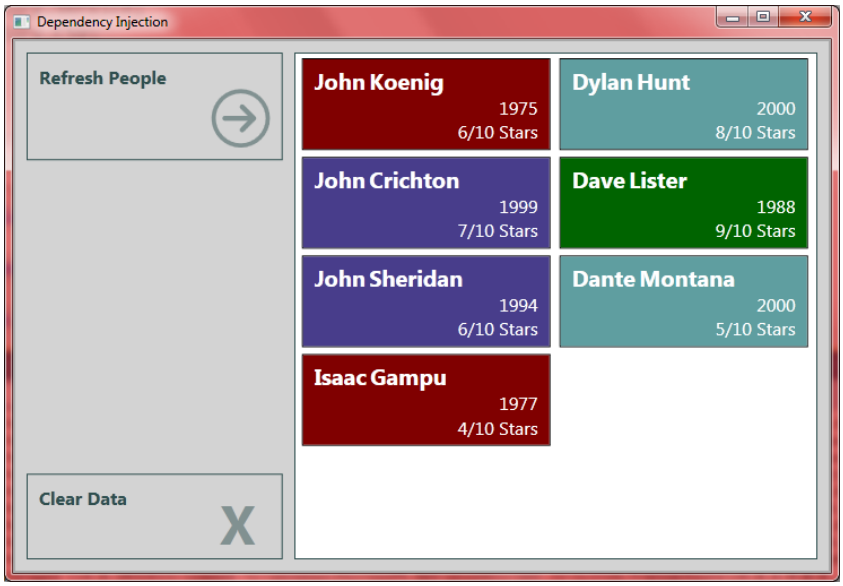
In composing our objects, first we instantiate the wrapped repository – in this case `PersonServiceRepository`. Then we instantiate our caching repository by passing it the instantiated `wrappedRepository`. Then we pass the caching repository to our view model just like we would pass any other repository instance.

And that is all the code we need to modify in our application. We do not need to modify the View, the View Model, or the underlying Repository. We simply compose our objects differently when injecting our dependencies.

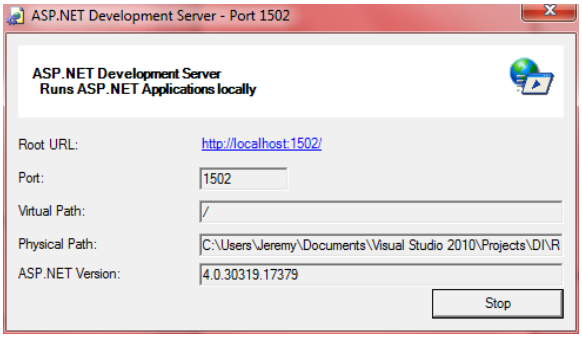
This is the power of Dependency Injection.

### *Testing the Cache*

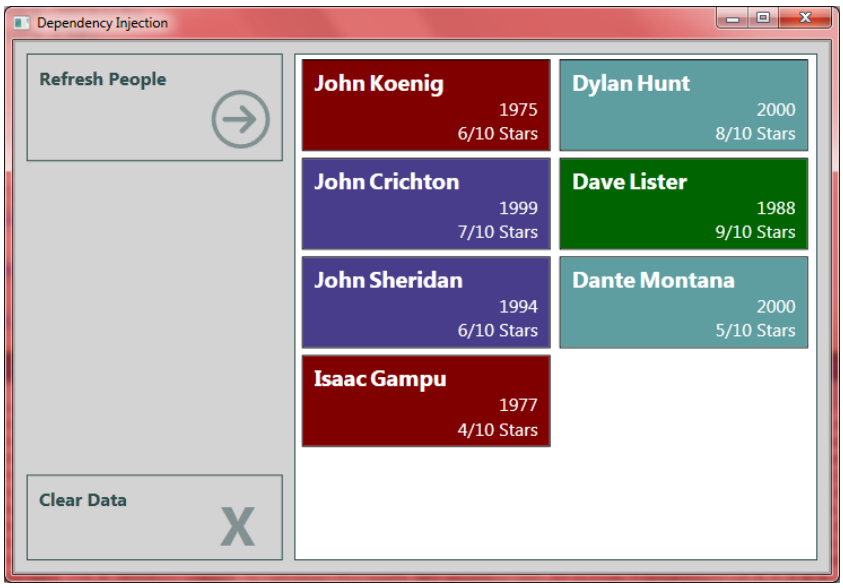
So, let's try out the code. If we run the application, we get the expected result:



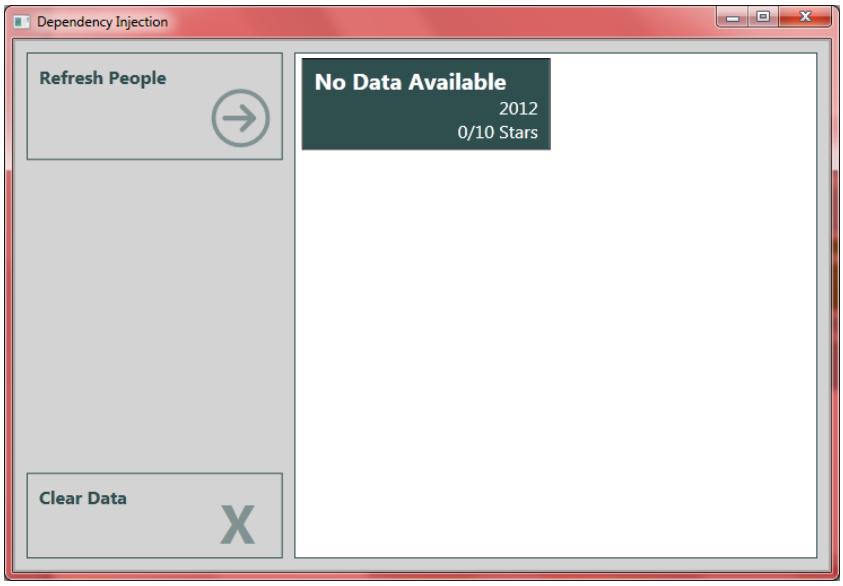

Now, here's where things get interesting. We're running the WCF service in the Cassini Server (the development server that comes with Visual Studio). Use Alt-Tab (or use the tray icon) to get to the ASP.NET Development Server:




Click the "Stop" button. This will shut down our WCF Service. Now, back in our application, click the "Clear Data" button to clear the list box, then click "Refresh People" again. If our cache is working (and less than 30 seconds have passed since our last refresh), we should see the following:




This shouldn't surprise us too much. After all, we're just hitting the client-side data cache. But if we wait 30 seconds, and try again, we get the following:




Since the WCF Service is not running, the `PersonServiceRepository` throws an exception when trying to connect to the service. This gets caught by the `CachingPersonRepository` and displayed as the record above.

What this has shown is that our client-side cache is working – it returns data even when the service is not available (at least until the cache expires).

## Unit Testing

So, we've seen how Dependency Injection can help us make our code more extensible and maintainable. But it can also help us make our code more testable. Unit Testing is a huge topic in itself. We'll just touch on a few points here.

As we mentioned regarding our first (non-DI) sample, it would be very difficult to unit test the View Model because we could not easily isolate the various parts. Since the View Model was tightly-coupled to a Repository, that repository would need to be included in the test as well. If the Repository failed, it could impact our test even though we weren't trying to test that part of the code.

But since we are now using DI to eliminate that tight-coupling, we have good "seams". This makes it extremely easy to create a mock of the dependency so that we can focus on testing the View Model code in isolation. Let's see exactly what that means.

### Testing the View Model

These test projects are using MSTest (which is included in all editions of Visual Studio 2012, and some (but not all) editions of Visual Studio 2010). If you do not have MSTest available, you won't be able to run these projects, but can still follow along with the code. The same principles apply to other unit test tools (such as NUnit).

Take a look at the `DI.Presentation.Test` project. Here, we have a unit test class for our View Model (`MainWindowViewModelTest`). If you're following along with the code, you will need to uncomment the methods in this class. Before we made changes to the View Model (by adding the constructor injection), these methods would not build.

```
[TestClass]
public class MainWindowViewModelTest
{
    IPersonRepository _repository;

    [TestInitialize]
    public void Setup()
    {
        var people = new List<Person>()
        {
            new Person() {FirstName = "John", LastName = "Smith"...,
            new Person() {FirstName = "Mary", LastName = "Thomas"...,
        };

        var repoMock = new Mock<IPersonRepository>();
        repoMock.Setup(r => r.GetPeople()).Returns(people);
        _repository = repoMock.Object;
    }

    [TestMethod]
    public void People_OnRefreshCommand_IsPopulated()...

    [TestMethod]
    public void People_OnClearCommand_IsEmpty()...
}
```

A quick overview of the tests: first, we have a class-level `IPersonRepository` variable (`_repository`). We are going to use this to hold our mock repository. A mock object is a placeholder object that acts as a stand in for a real object. Mock objects are usually created by some sort of mocking framework (such as Moq or RhinoMocks) and allow us to configure specific behavior, if we like.

Next, we have a `Setup` method. Since this is marked with the `TestInitialize` attribute, it will be run before each `TestMethod` is run. In the Setup, we first create a hard-coded collection of `Person` objects for our tests to use. Then we use the Moq framework to create a new mock object based on the `IPersonRepository` interface. We don't really have the space to go into details about mocking here. In short, this code is creating a mock repository and specifying that if someone calls the `GetPeople` method on the mock object, it should return our hard-coded `people` collection.

> *As a side note: I've been using Moq in my own unit testing and have had good success with it. (There are other good mocking tools available as well.) A couple things I like about Moq are the fluent interface and that it is contained in a single, bin-deployable assembly (Moq.dll). The Moq assembly is included in the "AdditionalAssemblies" folder in the code download. It is also available through NuGet.*

The last line in `Setup` assigns our mocked repository to our `IPersonRepository` variable. So, let's see how this is used in the tests themselves:

```
[TestMethod]
public void People_OnRefreshCommand_IsPopulated()
{
    // Arrange
    var vm = new MainWindowViewModel(_repository);

    // Act
    vm.RefreshPeopleCommand.Execute(null);

    // Assert
    Assert.IsNotNull(vm.People);
    Assert.AreEqual(2, vm.People.Count());
}
```

This test verifies that when the `RefreshPeopleCommand` is executed, the `People` property of the View Model is populated. The first step is to create an instance of our View Model. Notice that we are passing our mocked repository as the parameter for this – this is the "seam" that we mentioned earlier. The next step is to execute the `RefreshPeopleCommand`. The last step is to verify the results – by checking that `People` is not null and that it contains the two items expected from our test data.

Pay careful attention to what we are testing and what we are not testing. We want to test the operation of the `RefreshPeopleCommand` (code from the View Model). We do not want to test whether the Repository is working properly (at least not in this test). With Dependency Injection, we are able to isolate our code by injecting a mock repository instead of an actual one. This ensures that any exceptions or test failures will be related to the code we are trying to test and not due to some failure in an external dependency (such as a failed network connection).

Note: we do want to also test that all of our components work together (integration testing), but this is separate from unit testing (which is focused on testing isolated work units).

As an alternative to using a mocking tool, we can simply create a fake repository class.

```
public class FakeRepository : IPersonRepository
```

Then, in each method, we can return hard-coded values that our tests can use. This is a little more work, and we have more classes to manage. That's one reason mocking is so popular; it saves us from having to create special classes just for testing.

## Testing the Service Repository

In addition to testing the View Model, we also want to test our Repository. And in the case of the `PersonServiceRepository`, we want to try to isolate the Repository code from the Service code. We'll use Dependency Injection for this as well. But instead of using Constructor Injection, we'll use *Property Injection*.

### *Property Injection*

Property Injection is a little more difficult to grasp, so let's take a look at some code as explanation. Our `PersonServiceRepository` originally looked like this:

```
public class PersonServiceRepository
{
    public IPersonService ServiceProxy { get; set; }

    public PersonServiceRepository()
    {
        ServiceProxy = new PersonServiceClient();
    }
    ...other members removed
}
```

In this code, we have a public property for the Service, and the constructor is responsible for instantiating an instance of that service.

Our updated code looks like this:

```
public class PersonServiceRepository : IPersonRepository
{
    private IPersonService _serviceProxy;
    public IPersonService ServiceProxy
    {
        get
        {
            if (_serviceProxy == null)
                _serviceProxy = new PersonServiceClient();
            return _serviceProxy;
        }
        set
        {
            if (_serviceProxy == value)
                return;
            _serviceProxy = value;
        }
    }
    // Constructor has been deleted from the class
    ...other members hidden
}
```

We have removed the default constructor from the code. Instead of instantiating the Service in the constructor, we will instantiate it in the getter for the `ServiceProxy` property. Now why would we do this?

Let's consider two scenarios. Scenario A: We instantiate the Repository and then start calling the methods (`GetPeople`, etc.). When we call a method, it will use the `ServiceProxy` property. Since the backing field is null during the first call, it will automatically instantiate a `PersonServiceClient` for us to use. This is the default behavior (and the behavior that we want in production runs).

Scenario B: We instantiate the Repository and then set the `ServiceProxy` property *before* calling any of the methods. By setting the property, we are injecting a dependency into the Repository. When we call the methods, it will use the Service instance that we injected (rather than the default instance).

*Property Injection* is the injection of dependencies based on writable properties. This is useful when we have a valid default value. Whether the default is overridden is entirely up to the consumer of the class. If there is no valid default value, then we should use Constructor Injection or some other method to ensure that the dependency is always set.

### *Property Injection in Unit Tests*

Now that we have our Repository configured for injection of the Service (through the writeable property), we can use this as a "seam" for our tests.

Take a look at the `DI.Repository.Service.Test` project. Here, we have a unit test class for our View Model (`PersonServiceRepositoryTest`).

```
[TestClass]
public class PersonServiceRepositoryTest
{
    IPersonService _service;

    [TestInitialize]
    public void Setup()
    {
        var people = new List<Person>()
        {
            new Person() {FirstName = "John", LastName = "Smith"...,
            new Person() {FirstName = "Mary", LastName = "Thomas"...,
        };

        var svcMock = new Mock<IPersonService>();
        svcMock.Setup(s => s.GetPeople()).Returns(people.ToArray());
        svcMock.Setup(s => s.GetPerson(It.IsAny<string>())).Returns(
            (string n) => people.FirstOrDefault(p => p.LastName == n));
        _service = svcMock.Object;
    }

    [TestMethod]
    public void GetPeople_OnExecute_ReturnsPeople()...

    [TestMethod]
    public void GetPerson_OnExecuteWithValidValue_ReturnsPerson()...


    [TestMethod]
    public void GetPerson_OnExecuteWithInvalidValue_ReturnsNull()...
}
```

This setup is almost exactly the same as the setup for the View Model test. The difference is that we are mocking the Service (`IPersonService`) instead of the Repository. We then place that mocked service into the `_service` variable.

Here is a test method:

```
[TestMethod]
public void GetPeople_OnExecute_ReturnsPeople()
{
    // Arrange
    var repo = new PersonServiceRepository();
    repo.ServiceProxy = _service;

    // Act
    var output = repo.GetPeople();

    // Assert
    Assert.IsNotNull(output);
    Assert.AreEqual(2, output.Count());
}
```

Things are a little bit different here. Our first statement instantiates `PersonServiceRepository` (using the default constructor). Next, we inject our mocked service through the repository's `ServiceProxy` property. This means that any calls we make to the service will go through the mock that we generated in our test setup. The rest of this method calls the `GetPeople` method and validates the results.

So, by using Property Injection, we have added a "seam" to our code that makes it more easily testable. When our application runs, it will automatically pick up the default value (`PersonServiceClient`) that is supplied for the property. But when we test, we can use a mock of the Service to isolate the Repository code.

## Dependency Injection with a Container

So far, we have implemented Dependency Injection by wiring up our dependencies manually in our composition root. But the most common way of using Dependency Injection is with a third-party dependency injection container / framework. These offer numerous advantages over building our own dependency injection container – including lifetime management, configuration, and dependency resolution.

The good news is that there are quite a few DI containers available for free (and many are open-source as well). These include Unity, Castle Windsor, Ninject, Autofac, StructureMap, Spring.NET, and several others. They offer similar features, and making a selection may be dependent on other factors. As an example, the DI container in Spring.NET is part of a much larger framework, and this framework is a .NET version of the Spring framework for Java. This would be a good choice to ensure consistency across .NET and Java environments.

In our samples, we will be using Unity (from the Microsoft Patterns & Practices team) and Ninject. These are both available as free downloads and are bin-deployable (meaning, we just need to include the assemblies in our output folder).

For Unity, functionality is broken down into several assemblies; this allows us to deploy only the functions that we are using. In these samples, we will be using the base assembly (`Microsoft.Practices.Unity.dll`) and the configuration assembly (`Microsoft.Practices.Unity.Configuration.dll`).

Unity offers several ways of configuring the DI container, including configuration in code and XML configuration. Unity has a lot of features, and we'll only look at a few of them. I would encourage you to explore the Unity documentation to learn the details of what is available.

## Configuration in Code – Unity

First, we'll configure our Unity container in code. We'll see that this is very similar to our manual configuration, but we will get added benefits. Let's take a look at the `DI.UI.Container.Unity` project (be sure to set this as the StartUp Project if you're following along).

In this project, we have already added a reference to `Microsoft.Practices.Unity.dll` (again, this is available in the AdditionalAssemblies folder in the code download). So, let's take a look at `App.xaml.cs`.

First, we have an additional using statement:

```
using Microsoft.Practices.Unity;
```

Here is the `App` class:

```
    public partial class App : Application
    {
        IUnityContainer Container;

        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            ConfigureContainer();
            ComposeObjects();
            Application.Current.MainWindow.Show();
        }

        private void ConfigureContainer()
        {
          Container = new UnityContainer();
          Container.RegisterType<IPersonRepository, PersonServiceRepository>
                                  (new ContainerControlledLifetimeManager());
        }


        private void ComposeObjects()
        {
            Application.Current.MainWindow = Container.Resolve<MainWindow>();
        }
    }
```

We have added a class level variable for our Unity container (`Container`). In our `OnStartup` method, we have added a new call to `ConfigureContainer`. This method is responsible for configuring the Unity container with the types we will use for our dependencies.

In `ConfigureContainer`, we instantiate our Unity container. Then we call the `RegisterType` method to associate the `PersonServiceRepository` with the `IPersonRepository` interface. This lets the Unity container know that anytime we ask for an `IPersonRepository`, it should give us an instance of `PersonServiceRepository`. The parameter for this method is a lifetime manager.

Lifetime management is a big topic (and one of the primary reasons we use a third-party container as opposed to building our own). Lifetimes determine whether we get new instances each time we ask for a dependency, whether we get the same instance each time, or some other variation.

In this case we are using the Unity `ContainerControlledLifetimeManager`. This is the `Singleton` lifetime in Unity, meaning that each time we resolve the dependency, we will reuse the same (shared) instance each time. The default lifetime for Unity is a `Transient` lifetime. With a `Transient` lifetime, Unity would provide us with a new (non-shared) instance of the dependency each time we ask for it. There are other lifetimes as well. For a thorough discussion, refer to Mark Seemann's book mentioned above.

### *Resolving the Dependencies from the Container*

Now that we have our Unity container configured, we can let the container inject the dependencies for us. We do this by asking the container to resolve an object.

```
Application.Current.MainWindow = Container.Resolve<MainWindow>();
```

Compare this to the line of code from our manually-wired project:

```
Application.Current.MainWindow = new MainWindow(viewModel);
```

In the previous project, we instantiated the `MainWindow` class ourselves. But here, we ask the Unity container to resolve the `MainWindow` for us. This leads to a couple of questions.

How can we resolve a type (`MainWindow`) when we never registered that type with container? The answer is that if we ask Unity for a concrete type (and the container can figure out where to find that type), then it will instantiate the object for us automatically. Notice that our project contains references to our class libraries; this allows Unity to resolve concrete types from these assemblies.

What about the dependencies? When we instantiated `MainWindow` ourselves, we had to give it a `MainWindowViewModel` as a parameter. How does Unity deal with this? The answer is that when we ask Unity to resolve a type (such as `MainWindow`), it looks through all of the constructors for that type and looks for the constructor with the most parameters that it can figure out how to resolve.

In our case, `MainWindow` has a single constructor that takes a `MainWindowViewModel` parameter. Since `MainWindowViewModel` is a concrete type, Unity can figure out how to instantiate it as well. When Unity looks at `MainWindowViewModel`, it finds the constructor that takes an

`IPersonRepository`. Since we registered `PersonServiceRepository` during configuration, the container knows what type to instantiate for this dependency. When the container instantiates `PersonServiceRepository`, it finds only the default constructor. Unity automatically wires all of these dependencies together for us when we make a request to `Resolve` the `MainWindow`.

When we run the application, we get the expected results that we've seen before.

As you can imagine, if we want to swap out the `PersonCSVRepository` for the `PersonServiceRepository`, we just need to change the `RegisterType` method call in the `ConfigureContainer` method.

## Configuration in Code – Ninject

The good news about the various Dependency Injection containers is that offer similar features. As another example, let's switch to the `DI.UI.Container.Ninject` project (be sure to set this as the Startup project if you're following along). To use Ninject, we follow the same steps as for Unity: we add the assembly reference and then add the `using` statement in our `App.xaml.cs`.

```
using Ninject;
```

Here's what our `App.xaml.cs` for the Ninject project:

```
public partial class App : Application
{
    IKernel Container;

    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        ConfigureContainer();
        ComposeObjects();
        Application.Current.MainWindow.Show();
    }

    private void ConfigureContainer()
    {
        Container = new StandardKernel();
        Container.Bind<IPersonRepository>().To<PersonServiceRepository>()
          .InSingletonScope();
    }

    private void ComposeObjects()
    {
        Application.Current.MainWindow = Container.Get<MainWindow>();
    }
}
```

The syntax is a little different. Our container type is `IKernel`. And when we instantiate this, we use a `StandardKernel`. These are both included in the Ninject assembly.

When we look at the configuration, we see that Ninject uses a very fluent syntax (meaning that we use the API by stringing methods together). We call `Bind` to select the abstraction (our

`IPersonRepository` interface). Then call `To` to specify the concrete type (`PersonServiceRepository`). Finally, we call `InSingletonScope` which sets the lifetime for the object.

When we want to get objects out of the container, we just use `Get` and specify the type (very similar to the `Resolve` of the Unity container).

## Late Binding and XML Configuration with Unity

Imagine that our application is deployed to multiple client sites. One of our clients would like to have a repository that uses an Oracle database. Wouldn't it be great if we could ship that client just the new repository assembly without needing to recompile the entire application? That's exactly what late binding gives us.

By using XML configuration with Unity, we can specify which types to load from specific assemblies. Then all we need to do in order to deploy a new repository is to ship the repository assembly and update the configuration file. The core application remains unchanged.

Take a look at the `DI.UI.LateBinding` project (be sure to set it as the StartUp Project). Here is the code in `App.xaml.cs`. First we have references to two Unity assemblies:

```
using Microsoft.Practices.Unity;
using Microsoft.Practices.Unity.Configuration;
```

As mentioned earlier, Unity ships functionality in separate assemblies. The XML configuration functionality is in the `.Configuration` assembly (which is added in our project references as well).

Here's the rest of the `App` class:

```
public partial class App : Application
{
    IUnityContainer Container;

    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        ConfigureContainer();
        ComposeObjects();
        Application.Current.MainWindow.Show();
    }

    private void ConfigureContainer()
    {
        Container = new UnityContainer();
        Container.LoadConfiguration();
    }

    private void ComposeObjects()
    {
        Application.Current.MainWindow = Container.Resolve<MainWindow>();
    }
}
```

Notice that instead of registering types in our `ConfigureContainer` method, we make a call to `LoadConfiguration`. This will load the Unity configuration from the `App.config` file. (Note: Unity only allows configuration to be loaded from `App.config` and not from an arbitrary XML file.)

Configuration consists of two parts. First, the configSection:

```
  <configSections>
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration"/>
  </configSections>
```

Then the configuration itself:

```
  <unity>
    <namespace name="DI.Common" />
    <namespace name="DI.Repository.Service" />
    <assembly name="DI.Common" />
    <assembly name="DI.Repository.Service" />
    <container>
      <register type="IPersonRepository" mapTo="PersonServiceRepository">
        <lifetime type="ContainerControlledLifetimeManager" />
      </register>
    </container>
  </unity>
```

First, we let Unity know what assemblies and namespaces we are using. We specify two different assemblies/namespaces here since the interface comes from one (`DI.Common`) and the concrete type comes from another (`DI.Repository.Service`).

Next, we have our `register` element. Just like when we configured the container in code, this associates the concrete type `PersonServiceRepository` with the interface `IPersonRepository`. Notice that we can also configure the lifetime manager here.

This results in truly dynamic loading of these assemblies and types. Note that this project does *not* have any reference to the `DI.Repository.Service` assembly; it is not included in the project references. We just need to make sure that Unity can locate this assembly. In our case, we copy the dll into the appropriate bin folder of our project – this is done through a post-build event on the `DI.Repository.Service` project for this sample. But we just need to copy the dll over to our output folder.

When we run the application, we get the same results as before. If we want to change to the `PersonCSVRepository`, we just need to update the configuration file and make sure that the `DI.Repository.CSV.dll` is in the bin folder. (The sample code already contains the assembly and namespace references in configuration to make this easier to change.)

### *Ninject and XML Configuration*

A quick note about Ninject: it does not offer an XML configuration option. One of the goals of the Ninject project is to free the developer from XML configuration. So, this option is not available.

### *Pros and Cons of XML Configuration*

We've already seen the advantages of XML configuration – it allows us to do late binding. We can update the functionality of our application by simply changing configuration. There is no need to recompile. But this comes at a cost.

The cost, in this case, is brittleness. We don't get any compile-time checks of our container configuration. This means that if we have any errors in the XML (such as a mistyped assembly name or interface), we will get runtime errors when the types are being resolved.

In contrast, when we configure the container in code, we get the compile-time checks that would prevent these types of errors, but we lose the late-binding capability.

The good news is that we can mix and match our configuration methods. A recommended approach is that we use XML configuration only for those elements for which we need late binding. For everything else, we can put the configuration in code to take advantage of the compile-time checks.

## Unit Testing with a Container

Unit testing with a DI container is very similar to the unit testing that we saw earlier. The difference is that in our setup method, we will configure our DI container; and in our tests, we will `Resolve` objects from the container rather than instantiating them ourselves.

Let's go back to the `DI.Presentation.Test` project and take a look at `ContainerMainWindowViewModelTest.cs`. As you can imagine, we need to add the Unity assemblies to our project references (and as a `using` statement in the test class). Here is our setup code:

```
IUnityContainer _container;

[TestInitialize]
public void Setup()
{
    _container = new UnityContainer();

    var people = new List<Person>()
    {
        new Person() {FirstName = "John", LastName = "Smith"...,
        new Person() {FirstName = "Mary", LastName = "Thomas"...,
    };

    var repoMock = new Mock<IPersonRepository>();
    repoMock.Setup(r => r.GetPeople()).Returns(people);
    // Once we have our mock, we register it with the DI container
    _container.RegisterInstance<IPersonRepository>(repoMock.Object);
}
```

This code is almost the same as our non-container test. The difference is that instead of having an `IPersonRepository` variable that we use for dependencies, we have an `IUnityContainer`. After we go through the mocking process, we register the instance with the Unity container.

`RegisterInstance` is a bit different from what we did in our application code. In this case, we have an already-instantiated type (our mock repository object) that we want to use for our dependencies. When we register an instance with the Unity container, we are telling the container to return this particular instance whenever we ask for the `IPersonRepository` interface.

Our test code is a little different as well:

```
[TestMethod]
public void UnityPeople_OnRefreshCommand_IsPopulated()
{
    // Arrange
    var vm = _container.Resolve<MainWindowViewModel>();

    // Act
    vm.RefreshPeopleCommand.Execute(null);

    // Assert
    Assert.IsNotNull(vm.People);
    Assert.AreEqual(2, vm.People.Count());
}
```

Instead of instantiating `MainWindowViewModel` with a "new" statement, we ask the container for an instance. As noted earlier, Unity will look through the constructors, find one with an `IPersonRepository` dependency, and then inject our mock repository into the constructor. So the result is the same, we end up with an instance of the `MainWindowViewModel` with a mock repository injected.

### *Property Injection with Unity*

So, we've seen how Unity handles constructor injection, but what about Property Injection? If you remember, we have a test for our Repository that injects the Service through a writable property. We can do this with Unity as well.

Let's take a look at `ContainerPersonServiceRepositoryTest.cs`. First the setup:

```
IUnityContainer _container;

[TestInitialize]
public void Setup()
{
    _container = new UnityContainer();


    var people = new List<Person>()
    {
        new Person() {FirstName = "John", LastName = "Smith"...,
        new Person() {FirstName = "Mary", LastName = "Thomas"...,
    };

    var svcMock = new Mock<IPersonService>();
    svcMock.Setup(s => s.GetPeople()).Returns(people.ToArray());
```

```
            _container.RegisterInstance<IPersonService>(svcMock.Object);
            _container.RegisterType<PersonServiceRepository>(
                 new InjectionProperty("ServiceProxy"));
        }
```

Just like before, we mock up the Service object. But notice how we configure our Unity container. The call to `RegisterInstance` will associate our mock service with the `IPersonService` interface (very similar to our last test).

The call to `RegisterType` is a little different. Here we are specifying that we want to explicitly register the `PersonServiceRepository` type. This method accepts a parameter array of `InjectionMember` – ways of giving Unity specific instructions about how to inject dependencies. In this case, we are using `InjectionProperty` (to specify that we want to use Property Injection). The parameter is the name of the parameter that should be injected (`ServiceProxy`).

This allows us to write the following test code:

```
        [TestMethod]
        public void UnityGetPeople_OnExecute_ReturnsPeople()
        {
            var repo = _container.Resolve<PersonServiceRepository>();

            var output = repo.GetPeople();

            Assert.IsNotNull(output);
            Assert.AreEqual(2, output.Count());
        }
```

Notice that we ask the container to provide us with an instance of `PersonServiceRepository`. The container will check its configuration and see that we have specified that the `ServiceProxy` property should be injected. Since `ServiceProxy` is an `IPersonService`, Unity will assign our configured mock service as the value of `ServiceProxy`.

## Wrap Up

We've seen a lot of samples and have gone through a lot of DI concepts. We have seen how we can use Dependency Injection to break tightly-coupled components into isolated, loosely-coupled components. We have used DI to add seams to our code to improve testability. We have seen a number of Dependency Injection patterns such as Constructor Injection and Property Injection. We have looked at wiring up dependencies in a Composition Root and how to use a DI container to resolve dependencies in our object graph.

We've also seen how Dependency Injection can help us adhere to the S.O.L.I.D. principles. These are principles of good object-oriented design, and as with other design principles, they have been shown to lead to better code in many situations. The one we didn't mention (the "I") is the Interface Segregation Principle. This principle has to do with how we split up our methods into different interfaces so that classes only have dependencies on methods they use. It's a good principle, but we didn't run across the

need to talk about it in our sample code. As mentioned in the introduction, the S.O.L.I.D. principles are a good topic to explore further.

After all of this, we have barely scratched the surface of Dependency Injection. Hopefully, this has given you a good taste of the subject as well as a practical guide of how DI can be used in real code. I would highly encourage further exploration on the topic. As mentioned, *Dependency Injection in .NET* by Mark Seemann is a good place to get more comprehensive information. There are more patterns to learn, more complex scenarios to explore, more "gotchas" to watch out for.

Dependency Injection is an extremely powerful set of patterns. It is not necessarily appropriate for every situation. But, as always, more tools in our toolbox make us more versatile developers.

Happy coding!