IEnumerable, ISaveable, IDontGetIt: Interfaces in .NET

An overview of interfaces and abstraction by JeremyBytes.com

Overview

Interfaces in .NET allow us to program at an appropriate level of abstraction. They allow us to create and use functionality in an object even if we do not know the object's concrete class type. With appropriate use of interfaces, we can build code that is extensible and loosely coupled.

First, we'll take a look at some definitions – what interfaces are and how they compare to abstract classes (and concrete classes). Then we'll take a quick look at an advantage to programming to an interface rather than to a concrete type (in this case, IEnumerable). Next, we'll create our own interface (an object repository) and create an extensible and testable application. Finally, we'll remove compile-time constraints by dynamically loading an implementation of our interface. So, let's get started!

What is an Interface?

The first thing we need to do is define what an interface is. Usually, when we hear "interface" we think about user interfaces – the controls and design that the user sees on the screen. But the interfaces we are looking at here are in the code – allowing us to specify functionality that is supported by a class.

From the Visual Studio documentation: "Interfaces describe a group of related functions that can belong to any class or struct."

The easiest way to think of an interface is as a contract. When a class implements an interface, it is committing to implementing a specific set of functions that the programmer can rely on. Let's take a look at some samples to help us understand this better.

A Simple Sample

We'll start with a completed console application that contains samples of a concrete class, an abstract class, and an interface. You can download the source code for the application here: http://www.jeremybytes.com/Downloads.aspx. The sample code we'll be looking at is built using .NET 4 and Visual Studio 2010 (however, everything will work with .NET 3.5 and Visual Studio 2008). The download includes several different solutions (some of which share projects). Each solution exists as a starter application (if you want to follow along) and the completed code.

We'll start by taking a look at the `SimpleSample.sln` solution.

A Concrete Class

The `Polygons` project contains several different objects. We'll start by look at the `ConcreteRegularPolygon` class.

```
public class ConcreteRegularPolygon
{
    public int NumSides { get; set; }
    public int SideLength { get; set; }

    public virtual double GetArea()
    {
        throw new NotImplementedException();
    }

    public double GetPerimeter()
    {
        return SideLength * NumSides;
    }

    public ConcreteRegularPolygon(int numSides, int sideLength)
    {
        NumSides = numSides;
        SideLength = sideLength;
    }
}
```

This is a standard class that contains 2 properties, 2 methods, and a constructor. This is a simple object that represents a polygon that has the same length for each side and contains as many sides as we like. One thing to notice is that we are using automatic properties (for NumSides and SideLength). The syntax that we see above is shorthand for a fully implemented property such as this:

```
private int _numSides;
public int NumSides
{
    get { return _numSides; }
    set { _numSides = value; }
}
```

In this sample, we have a private field (_numSides) that holds the actual value. The getter and setter of the property (NumSides) interacts with the field.

> *Note: We may want to use the full property syntax if we need to add additional logic to the get or set (such as data validation or change notification). If we don't need to add code, then we can use the automatic property syntax to make our code a bit more compact.*

The `GetPerimeter` method of our class simply multiplies the length of each side by the total number of sides; this gives us the distance around the polygon. Notice that the `GetArea` method throws a `NotImplementedException`. This is because the calculation for the area of a polygon depends on the number of sides; each shape has a different formula. We've made this method `virtual` and expect that a descendent class will override this method to provide a useful implementation. We'll see a sample of this in use in just a bit.

An Abstract Class

An abstract class is a class that has one or more elements marked as `abstract` (nice use of recursion, huh?). An abstract method or property is just declared in the class; it is not actually implemented. Because of this, an abstract class cannot be instantiated. You must create a descendent class that implements the abstract members before an object can be instantiated in the code.

Here is the `AbstractRegularPolygon` from the sample project:

```
public abstract class AbstractRegularPolygon
{
    public int NumSides { get; set; }
    public int SideLength { get; set; }

    public abstract double GetArea();

    public double GetPerimeter()
    {
        return SideLength * NumSides;
    }

    public AbstractRegularPolygon(int numSides, int sideLength)
    {
        NumSides = numSides;
        SideLength = sideLength;
    }
}
```

Like our concrete class, this class contains 2 properties, 2 methods, and a constructor. But there are a couple of differences. First, the class itself is marked as `abstract`. Next, the `GetArea` method is also marked as `abstract` and does not contain an implementation. Because of this, we cannot instantiate the `AbstractRegularPolygon` class. We will need to create a descendent class that implements this method. We'll see this in just a bit.

A purely abstract class is a class that does not have any implementation code at all, only declarations. You don't meet purely abstract classes in the real world; an interface is usually a better choice.

An Interface

An interface is similar to a purely abstract class, in that it contains only declarations and no implementation code. But there are some key items that differentiate abstract classes and interfaces.

The sample project contains an `IRegularPolygon` interface.

```
public interface IRegularPolygon
{
    int NumSides { get; set; }
    int SideLength { get; set; }

    double GetArea();
    double GetPerimeter();
}
```

The interface contains 2 properties and 2 methods. Here are a few things to note. First, we use the `interface` keyword (instead of class or struct). Next, there are no access modifiers on the properties or methods. For interfaces, all members are public. This makes sense; since an interface is a contract between a class and a programmer, the members of that contract must be visible to both parties. Finally, there is no constructor.

> *Important note: we have a bit of unfortunate syntax here. If you look at the properties of the interface, they look very similar to the automatic properties that we have in the classes. But they aren't. The properties in an interface are **only** declarations, **not** implementations. So, although these look like automatic properties, they are really just a placeholder. The properties must be implemented in the class in order to fulfill the contract (as we'll see in just a bit).*

Class Implementation

Now that we've seen the base classes and the interface, let's take a look at using these items. The implementation is in the `Polygons.ConsoleApp` project.

Concrete Class Implementation

First, the `ConcreteSquare` descends from the concrete class:

```
class ConcreteSquare : ConcreteRegularPolygon
{
    public ConcreteSquare(int sideLength) :
        base(4, sideLength) { }

    public override double GetArea()
    {
        return SideLength * SideLength;
    }
}
```

The implementation is pretty straight-forward. This class descends from `ConcreteRegularPolygon`. The constructor simply calls the base class constructor, and there is an `override` of the `GetArea` method to implement the code. The rest of the implementation comes from the base class. No surprises here.

It is important to note that there is no requirement that we implement the `GetArea` method; if we wanted, we could remove this method and rely on the base class implementation (which throws the exception). If we remove the `GetArea` method, our code will still compile.

Abstract Class Implementation

Next, the `AbstractTriangle` descends from the abstract class. This implementation is similar to the previous class:

```
public class AbstractTriangle : AbstractRegularPolygon
{
    public AbstractTriangle(int sideLength) :
        base(3, sideLength) { }

    public override double GetArea()
    {
        return SideLength * SideLength * Math.Sqrt(3) / 4;
    }
}
```

The class descends from `AbstractRegularPolygon`. The constructor calls the base class constructor, and there is an `override` of the `GetArea` method. The rest of the implementation (the properties and `GetPerimeter`) comes from the base class.

The difference here is that we *must* implement the `GetArea` method. If we do not implement this method, the compiler will throw an error.

Interface Implementation

Finally, the `InterfaceOctagon` implements the interface:

```
public class InterfaceOctagon : IRegularPolygon
{
    public InterfaceOctagon(int sideLength)
    {
        NumSides = 8;
        SideLength = sideLength;
    }

    public int NumSides { get; set; }
    public int SideLength { get; set; }

    public double GetArea()
    {
        return SideLength * SideLength * (2 + 2 * Math.Sqrt(2));
    }

    public double GetPerimeter()
    {
        return SideLength * NumSides;
    }
}
```

First, I want to note the terminology. `InterfaceOctagon` *implements* the `IRegularPolygon` interface; it does not *descend* from the interface. Since the base class is not specified, the class implicitly descends from `Object`.

When we look at the code, we see a fully implemented constructor (because there is no base constructor). We also see implementations for both of the properties and for both of the methods.

> *Note again the unfortunate syntax. Even though the properties look just like the ones in the interface, these are actually automatic properties (like we saw in our concrete class, above). This*

qualifies them as an implementation of those properties, and as such, fulfills the interface contract.

If we do not provide implementations for all 4 members, then we will get a compiler error. Again, think of the interface as a contract between the class and the programmer – by implementing an interface, the class commits to having each of those members. A bit later, we'll take a look at some shortcuts that Visual Studio provides to make implementing interfaces easier.

What's the Difference?

So here's the big question: what's the difference between a purely abstract class and an interface? On the surface, they seem to both behave the same. However, there are key differences:

Abstract Classes	Interfaces
My contain implementation code (non-abstract members).	May not contain any implementation code. Interfaces are limited to declarations only.
A class may only descend from a single base class (single inheritance).	A class may implement multiple interfaces.
Members contain access modifiers (public, private, protected, etc.).	Members do not contain access modifiers. All members are automatically public.
May contain fields, properties, constructors, destructors, methods, events, and indexers.	May contain properties, methods, events, and indexers. May not contain fields, constructors, or destructors.

C# allows for single-inheritance only; meaning, that a class can descend from one (and only one) base class. There is no way for a class to descend from 2 different parent classes. Some people have cited this as a shortcoming in C# languages (after all, C++ allows for multiple-inheritance). But in practical usage, multiple-inheritance can lead to unanticipated issues (but that's a topic for another discussion).

The solution is interfaces. A class can descend from only a single base class, but it can implement as many interfaces as it needs. If we take a look at some of the classes in the .NET help system, we will see many examples of classes that implement multiple interfaces. As an example, here is the definition for `List<T>`:

```
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IList, ICollection, IEnumerable
```

In this case, there is no declared base type (so the base type is `Object`), but there are several generic and non-generic interfaces that are implemented. This means that when we use the `List<T>` class, we can depend on it having `IEnumerable` functionality. This will be important to us in our next example.

Programming to an Abstraction

Among the extensive list of best practices, we often hear that we should program to an abstraction rather than a concrete implementation. In order to see what this means and why it's important, let's take a look at another solution: `IEnumerableSample.sln`.

The Projects

The solution contains 3 projects: the `People.Service` project defines a WCF service that provides our data; we will be using the `GetPeople` method. The `PersonRepository.Interface` project will be more important in the next solution; for now, it's just used because it contains the declaration of the `Person` class (the class type of our sample data). We'll be spending most of our time in the `IEnumerable.UI` project.

The `IEnumerable.UI` project is a simple WPF application. The `MainWindow.xaml` contains a list box that will hold our data and 3 buttons – 2 to fetch data and 1 to clear the list box.

The project also contains a `Converters.cs` file that contains a number of value converters that are used in the display. If you are interested in the XAML and value converters in this project, you can look up the *Introduction to Data Templates and Value Converters in Silverlight* demo and sample code on the website: http://www.jeremybytes.com/Downloads.aspx (note: this works perfectly well in WPF as well).

The code-behind the `MainWindow` has stubs for the button event handlers. We will be filling these in shortly.

> *Note: We are not doing a fully layered / abstracted application here in order to keep the example simple and allow us to focus on the core parts. In a production application, the recommendation is to separate the model, application logic, and UI (such as is common when implementing the Model-View-ViewModel pattern).*

Adding the Service Reference

We'll start by adding a reference to the WCF Service. Here are the steps:

1. In the Solution Explorer, right-click on the project `IEnumerable.UI`, and select "Add Service Reference".
2. In the dialog, click the "Discover" button. This will look for any services that are included in the solution. `PersonService.svc` should come up.
3. In the "Namespace" box, type "MyService".
4. While we're on this screen, take note of the "Advanced" button. Don't click it now; we'll be looking it at later on.
5. Click the "OK" button.

The Concrete Implementation

At the top of the `MainWindow.xaml.cs` file, add the following using statement:

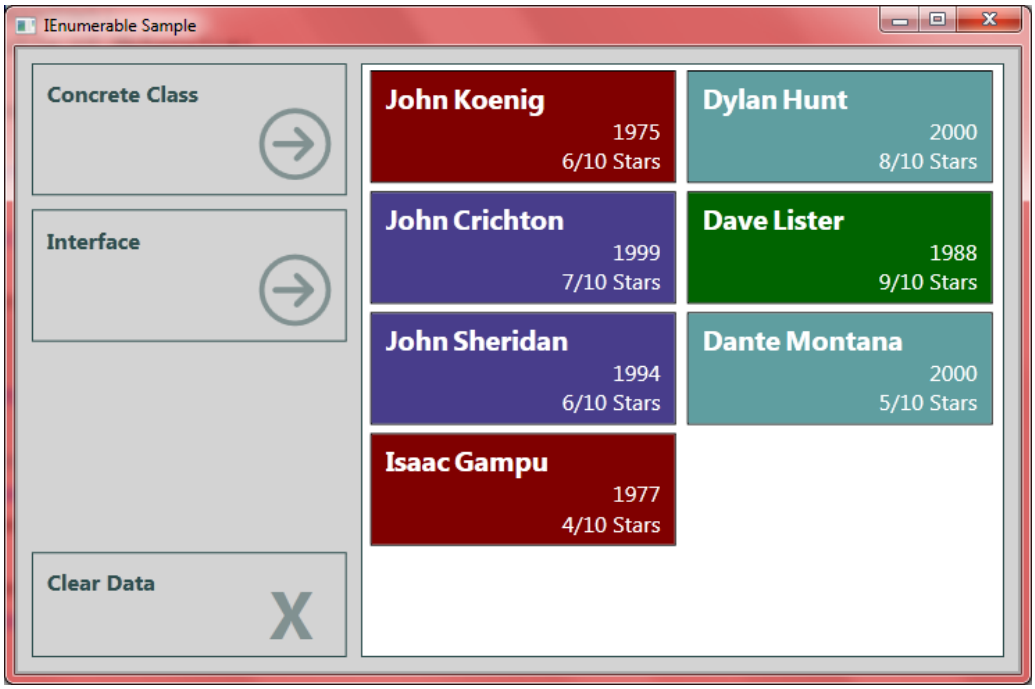
```
using IEnumerable.UI.MyService;
```

We'll start by implementing the `ConcreteFetchButton_Click` method. The idea behind this method is that we are coding to a concrete class (the class type that is returned by the WCF service). We'll use this to populate the list box. Here's the code:

```
private void ConcreteFetchButton_Click(object sender, RoutedEventArgs e)
{
    Person[] people;
    var proxy = new PersonServiceClient();
    people = proxy.GetPeople();

    PersonListBox.Items.Clear();
    foreach (var person in people)
        PersonListBox.Items.Add(person);
}
```

In this case, we create a variable which is an array of `Person`. This is the type that is returned by the `GetPeople` method of the service. Then we create an instance of the service and populate our variable. Finally, we loop through the items and add them to the list box. Here is the running application after clicking the "Concrete Class" button:



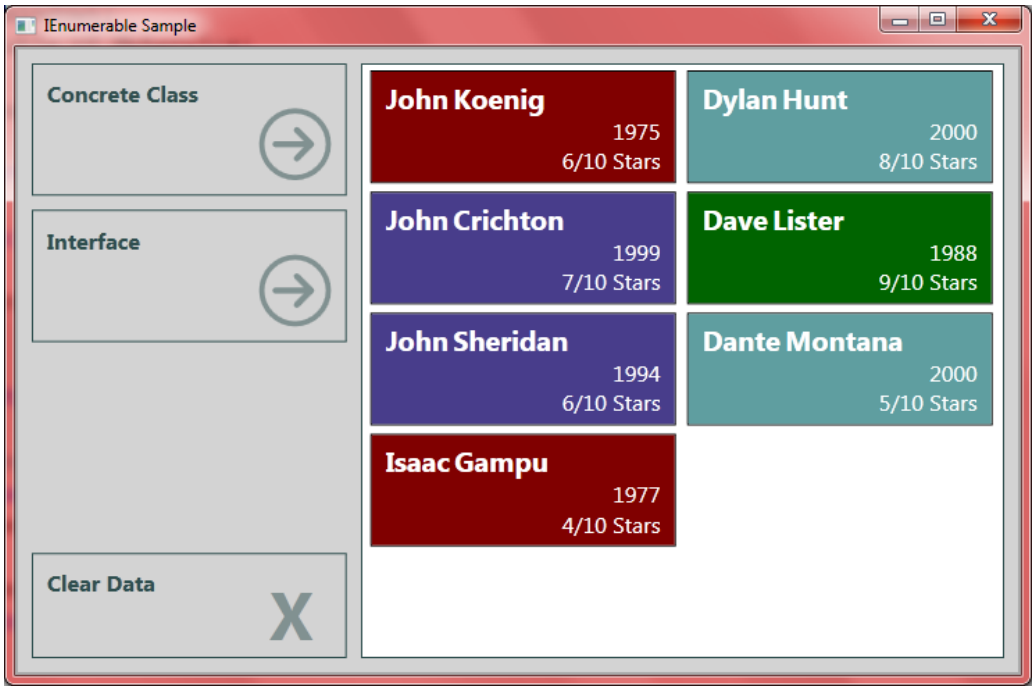
The Abstract Implementation

Now that we've seen the concrete implementation, let's take a step back and think about what we really need here. We have a collection of `Person` objects, and we are using a `foreach` loop to add them to a list box. But it turns out that the actual type of the collection doesn't really matter to us. All we really care about is the ability to use `foreach`. If we check the documentation, we find that in order to use `foreach`, we simply need a class that implements `IEnumerable` or `IEnumerable<T>`. So, for the `InterfaceFetchButton_Click` method, we'll program to the abstraction (the interface) rather than the concrete class:

```
private void InterfaceFetchButton_Click(object sender, RoutedEventArgs e)
{
    IEnumerable<Person> people;
    var proxy = new PersonServiceClient();
    people = proxy.GetPeople();

    PersonListBox.Items.Clear();
    foreach (var person in people)
        PersonListBox.Items.Add(person);
}
```

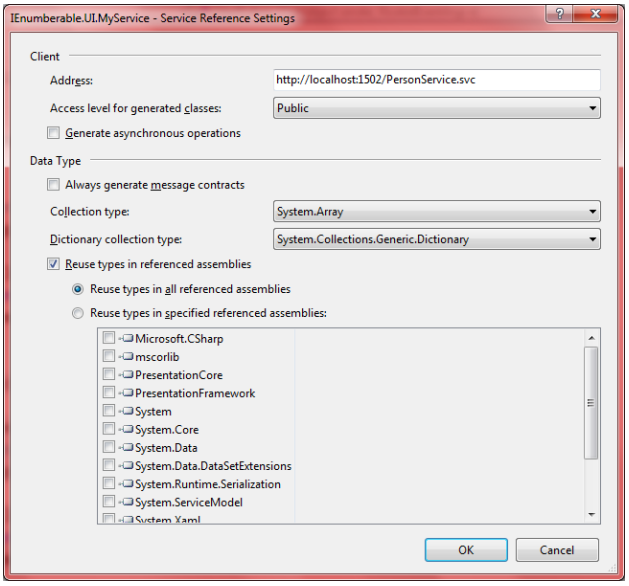
The only difference from the previous method is that our variable now specifies that interface (`IEnumerable<Person>`) rather than the concrete class (array of `Person`). If we run the application, we get exactly the same results:



So What?

Okay, so we've created 2 methods that have the same result. Why should we use the interface? For the same reason that we program to abstractions: to protect our code from change. Let's experiment a little with our application.

In the Solution Explorer, expand the Service References node, right-click on MyService, and select "Configure Service Reference."



Take a look at is the "Collection Type" drop-down. Right now, we can see that it is set to "System.Array", with the result that the service method returned an array (`Person[]`). This is the default type that we get when we bring in a service using BasicHttpBinding. This default type varies based on the service type and the programming environment that we are using. But this dialog allows us to change this. Let's update it to "System.Collections.Generic.List"; now our service method will return a `List<Person>`. As a side note, remember that "Advanced" button from the Add Service Reference dialog? That takes us to this same screen, so we can set these properties when we initially add the service to our project.

Now we have a problem. If we try to build the application, we get the following error:

```
Cannot implicitly convert type
'System.Collections.Generic.List<IEnumberable.UI.MyService.Person>' to
'IEnumberable.UI.MyService.Person[]
```

In order to get this application to build, we need to change the variable type in the `ConcreteFetchButton_Click` method from

```
Person[] people;
```

to

```
List<Person> people;
```

After making this change, our code will compile and run once more.

But the important thing to note is that we do **not** need to update the `InterfaceFetchButton_Click` method. Because we programmed to the abstraction (the `IEnumerable` interface), we are not affected by the change of the type coming back from the service. Since almost every collection in .NET implements the `IEnumerable<T>` interface, this section of code is robust and resistant to breaking in

response to this type of change. Granted, this is a very simple example, and there are other ways to resolve the issue in this scenario (such as with the use of `var`), but this gives us a good jumping-off point to look at some more realistic examples.

Using Interfaces in the Real World

Now that we've seen the usefulness of programming to an abstraction, let's take this further. In this case, we'll decouple our code by using the Repository pattern. The Repository pattern will allow us to separate the core of our application from the data storage mechanism.

> *Note: We are not doing a fully layered / abstracted application here in order to keep the example simple and allow us to focus on the core parts. In a production application, the recommendation is to separate the model, application logic, and UI (such as is common when implementing the Model-View-ViewModel pattern).*

In our previous example, we retrieved our data from a WCF service. But what if we wanted to expand this so that we could retrieve the data from a SQL Server or a CSV file (or any other data store that we can think of)? If we split things out a bit, we can make our application more extensible.

So let's take a look at the `RepositoryInterface.sln` solution.

The Projects

The initial solution contains 3 projects: the `People.Service` project is the same as from the previous example and contains a number of methods in a WCF service for us to use. The `PersonApp.UI` project is a WPF application that is very similar to the last application; it contains a list box with 3 buttons – one for each of the data storage types we want to use. The `PersonRepository.Interface` project contains the declaration of the `Person` class and also the repository interface that we are interested in. This is where we will start.

IPersonRepository

The `IPersonRepository.cs` file contains the declaration for our repository interface. This contains the methods that we may want to use to fetch and save data:

```
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);
}
```

These methods should be pretty self-explanatory. Our samples will be concentrating on `GetPeople` (which returns the entire list of `Person` objects) and `GetPerson` (which returns a single `Person` object based on last name).

As we noted earlier, an interface contains only declarations with no implementation. This particular interface only has methods, but properties, events, and indexers could be added if we needed them.

> *Further Exploration: This is an example of the Repository Pattern. This repository interface is tuned for a specific object type (`Person`). To make this more flexible, we could update our repository to use generics (`IRepository<T>`) and reflect this in the names/types of the methods.*

The WCF Service Repository

Now that we have an interface defined, we need to create a concrete class that implements this interface. We'll start with an implementation that uses the WCF Service. To do this, we'll create a new project in the solution:

1. In the Solution Explorer, right-click on the solution and select "Add", then "New Project".
2. Select "Class Library" as the template.
3. Name the project "PersonRepository.Service".
4. Right-click on the project, and select "Add Reference".
5. On the "Projects" tab, select "PersonRepository.Interface".
6. Right-click on the project, and select "Add Service Reference".
7. Follow the instructions from the previous sample to add the reference to the PersonService.
8. Rename the "Class1.cs" file to "ServiceRepository.cs". When prompted, select to rename the class as well.

Now that we have the basics set up, we can start modifying our code. Add the following to the top of the `ServiceRepository.cs` file:

```
using PersonRepository.Interface;
using PersonRepository.Service.MyService;
```

Add the interface to the class declaration:

```
public class ServiceRepository : IPersonRepository
{
}
```

If we try to build the application now, we will get errors because our class says that it implements `IPersonRepository`, but it doesn't actually implement any of the members. We could add the methods one at a time or copy them from the interface definition, but Visual Studio gives us a much easier way.

If click somewhere in the word "IPersonRepository" (to put the cursor there), you will see the first letter is underlined with a small blue box. This box is a clue that Visual Studio wants to help us out. If we

press the Ctrl key and ".", we get a pop-up menu. One of the items is "Implement interface 'IPersonRepository'". If we press enter, then Visual Studio stubs out all of the interface members for us. Note: we can get the same options by right-clicking on "IPersonRepository". There is also an option to explicitly implement the interface. This is a more advanced topic that is outside the scope of this document.

```
public class ServiceRepository : IPersonRepository
{
    public IEnumerable<Person> GetPeople()
    {
        throw new NotImplementedException();
    }

    public Person GetPerson(string lastName)
    {
        throw new NotImplementedException();
    }

    public void AddPerson(Person newPerson)
    {
        throw new NotImplementedException();
    }

    public void UpdatePerson(string lastName, Person updatedPerson)
    {
        throw new NotImplementedException();
    }

    public void DeletePerson(string lastName)
    {
        throw new NotImplementedException();
    }

    public void UpdatePeople(IEnumerable<Person> updatedPeople)
    {
        throw new NotImplementedException();
    }
```

This makes it extremely easy for us to make sure that we are implementing all of the required members of the interface. With that in mind we just need to replace the exceptions with our custom code. In this case, we are connecting to the WCF PersonService that implements these same methods. We'll add a constructor that creates the service proxy and then pass through all of the method calls to the service. Here's the completed code:

```
public class ServiceRepository : IPersonRepository
{
    private PersonServiceClient proxy;

    public ServiceRepository()
    {
        proxy = new PersonServiceClient();
    }
```

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

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

        public void AddPerson(Person newPerson)
        {
            proxy.AddPerson(newPerson);
        }

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

        public void DeletePerson(string lastName)
        {
            proxy.DeletePerson(lastName);
        }

        public void UpdatePeople(IEnumerable<Person> updatedPeople)
        {
            proxy.UpdatePeople(updatedPeople.ToArray());
        }
    }
```

Now we should be able to successfully build our solution. The next step will be to add our new class to the UI project. Let's do the following in the `PersonApp.UI` project.

1. In the Solution Explorer, right-click on "PersonApp.UI" and select "Add Reference".
2. On the "Projects" tab, select "PersonRepositoy.Service".
3. Right-click on "PersonApp.UI" and select "Add Service Reference".
4. Follow the instructions from the previous sample to add the reference to the PersonService.

Add the following to the top of the `MainWindow.xaml.cs` file:

```
using PersonRepository.Service;
```

Now we can implement the `ServiceFetchButton_Click` method:

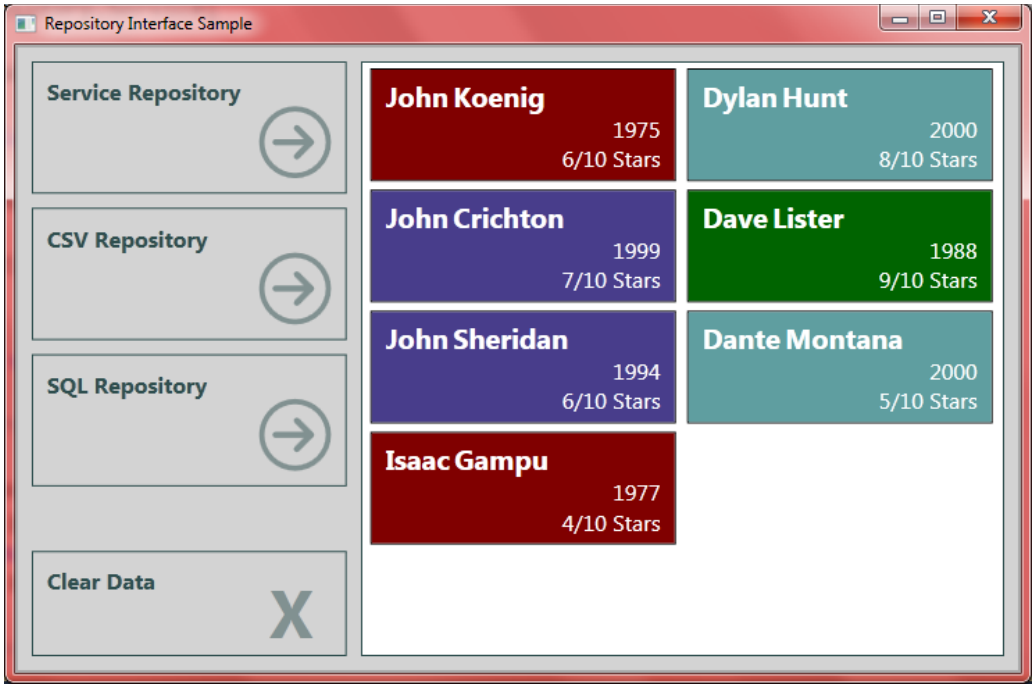
```
    private void ServiceFetchButton_Click(object sender, RoutedEventArgs e)
    {
        IPersonRepository repository = new ServiceRepository();

        PersonListBox.Items.Clear();
        var people = repository.GetPeople();
        foreach (var person in people)
            PersonListBox.Items.Add(person);
```

```
            MessageBox.Show(string.Format("Repository Type:\n{0}",
                repository.GetType().ToString()));
        }
```

Notice that we create a variable with the type `IPersonRepository`. This means that we can assign any object that implements that interface – in this case, the concrete type is `ServiceRepository`. The last statement (the message box) will show us the name of the concrete type. This will help us see some differences once we implement some more repository classes.

With this in place, we can run the application. Here is the result if we click the Service Repository button:



The CSV and SQL Repositories

Now that we have one repository implemented, we can take a look at the other two. The good thing is that these repository classes are already implemented; we just need to add them to the solution.

1. In the Solution Explorer, right-click on the "RepositoryInterface" solution, select "Add", then "Existing Project".
2. Locate "PersonRepository.CSV.csproj" and add it to the solution.
3. Repeat the above steps to add the "PersonRepository.SQL.csproj" project.
4. Right-click on the "PersonApp.UI" project and select "Add Reference".
5. On the "Projects" tab, select "PersonRepository.CSV" and "PersonRepository.SQL".

Add the following at the top of the `MainWindow.xaml.cs` file:

```
using PersonRepository.CSV;
using PersonRepository.SQL;
```

Add the following code for the `CSVFetchButton_Click` method:

```
private void CSVFetchButton_Click(object sender, RoutedEventArgs e)
{
    IPersonRepository repository = new CSVRepository();

    PersonListBox.Items.Clear();
    var people = repository.GetPeople();
    foreach (var person in people)
        PersonListBox.Items.Add(person);

    MessageBox.Show(string.Format("Repository Type:\n{0}",
        repository.GetType().ToString()));
}
```

And add the following code for the `SQLFetchButton_Click` method:

```
private void SQLFetchButton_Click(object sender, RoutedEventArgs e)
{
    IPersonRepository repository = new SQLRepository();

    PersonListBox.Items.Clear();
    var people = repository.GetPeople();
    foreach (var person in people)
        PersonListBox.Items.Add(person);

    MessageBox.Show(string.Format("Repository Type:\n{0}",
        repository.GetType().ToString()));
}
```

Notice that these methods are exactly the same as the method for our `ServiceRepository`. The only difference is the class that we "new" up in the first line. (This should also be a clue to us that we should consider refactoring this code; we'll do this in just a bit.)

The reason that we can use these classes so interchangeably is that they all implement the same interface. If we open up the `CSVRepository` class file and the `SQLRepository` class file, we will see that the implementations are wildly different. But since our application is programming to the interface (`IPersonRepository`), it is not concerned about these differences. The application only cares about the contract; because of the interface, the application is assured that each of these classes has a `GetPeople` method that can be called.

Now if we run the application, we should see the same results regardless of the button that is clicked. We can click the "Clear Data" button in between just to make sure that the list box is getting populated as expected. The MessageBox dialog that pops up shows the type of the repository that was used. This lets us verify that we are, in fact, getting results from three different repository types.

Refactoring

As mentioned above, we have a danger sign in our application: duplicated code. Let's start by creating a factory for our repository. With the Factory Method pattern, we pass the responsibility for instantiating

a particular type to another method. In this case, we'll create a static method that will create the proper repository class based on a parameter. We'll start by adding a new class:

1. In the Solution Explorer, right-click on the PersonApp.UI project, select "Add", then "Class".
2. Name the file "RepositoryFactory.cs".
3. Add the following using statements:

```
using PersonRepository.Interface;
using PersonRepository.Service;
using PersonRepository.CSV;
using PersonRepository.SQL;
```

Modify the class as follows:

```
public static class RepositoryFactory
{
    public static IPersonRepository GetRepository(string repositoryType)
    {
        IPersonRepository rep;

        switch (repositoryType)
            {
            case "Service": rep = new ServiceRepository();
                break;
            case "CSV": rep = new CSVRepository();
                break;
            case "SQL": rep = new SQLRepository();
                break;
            default:
                throw new ArgumentException("Invalid Repository Type");
        }
        return rep;
    }
}
```

Back in `MainPage.xaml.cs` we'll add a new method called "FetchData" to hold our common code:

```
    private void FetchData(string repositoryType)
    {
        IPersonRepository repository =
            RepositoryFactory.GetRepository(repositoryType);

        PersonListBox.Items.Clear();
        var people = repository.GetPeople();
        foreach (var person in people)
            PersonListBox.Items.Add(person);

        MessageBox.Show(string.Format("Repository Type:\n{0}",
            repository.GetType().ToString()));
    }
```

Now our button click methods become very simple:

```
private void ServiceFetchButton_Click(object sender, RoutedEventArgs e)
{
    FetchData("Service");
}

private void CSVFetchButton_Click(object sender, RoutedEventArgs e)
{
    FetchData("CSV");
}

private void SQLFetchButton_Click(object sender, RoutedEventArgs e)
{
    FetchData("SQL");
}
```

So, what we have now is a shared function (`FetchData`) which is referencing the `IPersonRepository` interface. Something important to note about this method is that it does not know anything about the concrete classes. All this method cares about is that the `GetPeople` method is implemented by the class, and that is guaranteed by the interface. The `FetchData` method makes a call out to the `RepositoryFactory` in order to get an instance of the repository. Again the local method doesn't care about the concrete type, only that it implements `IPersonRepository`.

You can see that our code looks very clean and that we could add additional repository types with very little effort. There are still a few shortcomings, but we'll get a little closer with the next example.

Late Binding – Making Decisions at Runtime

One problem with our current application is that it is statically linked. Our main project (PersonApp.UI) needs to have references to each of the repository classes that we want to use. This means that if we add a new repository, we need to recompile the application. We want to try to decouple this so that we can create and deploy new repositories without modifying the application.

To do this, we'll implement a low-budget version of the Inversion of Control (IoC) pattern. There's some confusion between Inversion of Control and Dependency Injection (or Dependency Inversion). In order to keep things simple, we'll just use the Wikipedia definition of Inversion of Control:

> In software engineering, Inversion of Control (IoC) is an object-oriented programming practice where the object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis.

Based on this definition, here are the basics of the IoC container that we will implement:

1. The application code only references the interface, never the concrete type.
2. When the application needs an instance of a class that implements the interface, it calls out to the IoC container and asks for it.

3. The IoC container then decides what class needs to be instantiated (through configuration or discovery) and then returns the class. This class type is not known at compile time; it is determined at runtime.
4. The application makes use of the instantiated class.

Note: Most fully-implemented IoC containers also contain some sort of lifetime management of the objects so that they are not constantly created and destroyed. For our example, we are not implementing this functionality.

We're going to update our RepositoryFactory so that it acts as our low-budget IoC container. It won't have much functionality, but it will get the job done. The factory will look in the configuration file for the type and assembly to use. It will then create an instance of that type and return it to the application.

Let's start by opening up the `DynamicLoadRepositoy.sln`.

The Projects

This solution contains 3 projects: the `PersonRepository.Interface` project is the interface that we have been dealing with. The `PersonApp.DynamicRepository.UI` project contains our UI and the factory. The UI is similar to previous samples except that it only contains 1 fetch button. The `People.Service` project has the WCF service. One important thing to note is that the UI project does *not* contain a service reference. The only reason the service is included in the solution is so that it will start up automatically when we run the application through Visual Studio.

Inversion of Control in Action

Let's start by looking at the `MainWindow.xaml.cs` file. This will look very similar to where we left the last project:

```
private void FetchButton_Click(object sender, RoutedEventArgs e)
{
    IPersonRepository repository = RepositoryFactory.GetRepository();

    PersonListBox.Items.Clear();
    var people = repository.GetPeople();
    foreach (var person in people)
        PersonListBox.Items.Add(person);

    MessageBox.Show(string.Format("Repository Type:\n{0}",
        repository.GetType().ToString()));
}
```

In the `RepositoryFactory.cs` file, we see that the factory method has not yet been implemented. Let's take a look at the implementation and then walk through the code:

```
public static IPersonRepository GetRepository()
{
    string typeName = ConfigurationManager.AppSettings["RepositoryType"];
    Type t = Type.GetType(typeName);
    object obj = Activator.CreateInstance(t);
    IPersonRepository rep = obj as IPersonRepository;
    return rep;
}
```

This code is a bit more advanced, so don't worry if you don't understand all of it right away. The first step is to get the fully-qualified type name from configuration (app.config). As a sample, the CSVRepository looks like this:

```
<add key="RepositoryType" value="PersonRepository.CSV.CSVRepository,
PersonRepository.CSV, Version=1.0.0.0, Culture=neutral"/>
```

After getting the fully qualified type name, we use it to generate a `Type` object. Now that we have a `Type` object, we can use the `Activator` class to instantiate a new object. The object generated by `CreateInstance` ultimately depends on the app.config. Based on the sample configuration, `CreateInstance` will return a CSVRepository instance. Notice here that we just assign it to a variable of type object. The next step is to cast this to `IPersonRepository` since that is the return type required by the method. The "`as`" statement attempts to cast the object to `IPersonRepository`; if it is not successful, then it simply returns null. The last step returns our `IPersonRepository` to the calling method.

This is a standard way of dynamically instantiating an object based on a type name. We could shorten this code a bit (by combining some of the calls and casting), but that makes it a bit more difficult to understand if you haven't worked with this type of code before.

There are a couple of other requirements in order for this code to work as expected. First, since the CSVRepository requires the path to the "People.txt" file, we have an entry for that in the configuration:

```
<add key="CSVFilePath" value="C:\Users\Jeremy\Documents\Visual Studio
2010\Projects\Interfaces\RunThrough\Interfaces\PersonApp.DynamicRepository.UI
\People.txt"/>
```

The only other requirement is that the "PersonRepository.CSV" assembly be placed where the application can find it. In this sample, we have a copy of "PersonRepository.CSV.dll" in the "bin" folder of our application; but we could also load the assembly into the GAC (Global Assembly Cache).

The sample project contains configuration file entries for the WCF service repository and the SQL repository. By updating the app.config file, we can determine which repository will be used at run time. The comments in the configuration file will show you how to use the other repositories. Nothing is statically hooked together at compile time. Instead, it is decoupled and dynamically loaded at run time.

This means that if we need to deploy another repository type, we just need to build the class library, put it into the application folder, and update the application configuration file (PersonApp.DynamicRepository.UI.exe.config). There is no need to recompile the application itself.

Late Binding

This is an example of late binding. We are making the decision of what concrete type we use at runtime rather than compile time. There are both advantages and disadvantages to this approach. The big advantage that we see here is that we can deploy new Repositories without needing to recompile the application – we just need to supply the appropriate assembly and update configuration. The disadvantage is that we lose the compile time error checking. If we try to configure an invalid assembly or the type in the assembly does not implement the expected interface, we only find out about this at runtime.

We have good decoupling of our application; we have eliminated the need for the application to know about all of the different repository types (this is good). To mitigate the issue of runtime errors we can make sure that we properly test our various repositories. If each of the repositories is unit-tested, then we know that the repository itself should behave as expected. When we add the seams to our code that create these separations, then the unit tests become much easier to write.

A Generic IoC Container

Our factory method is now much more flexible since it dynamically loads our repository. But it is limited to a single type: `IPersonRepository`. A useful IoC Container will be able to return a variety of types. We can create a container class with generics to make this possible. We'll add this class to the `RepositoryFactory.cs` file (just for convenience) as a sibling to the `RepositoryFactory` class:

```
public static class Container
{
    public static T Resolve<T>() where T : class
    {
        string configString =
            ConfigurationManager.AppSettings[typeof(T).ToString()];
        Type resolvedType = Type.GetType(configString);
        return Activator.CreateInstance(resolvedType) as T;
    }
}
```

This code is similar to our factory method. The primary difference is that rather than having `IPersonRepository` hard-coded as the type, it uses generics to work with whatever type we want. Using the generic type, it gets a setting out of configuration (we'll see that setting in just a bit). Then it goes through the same steps by resolving the `Type` and then using the `Activator` to create an instance. You can see that we combined the last 3 lines from our previous method into a single statement that creates the instance, casts it to the appropriate type, and then returns it to the calling method.

We have to make a few changes to use this new `Container` class. Back in our MainPage.xaml.cs, we update the call from

```
IPersonRepository repository = RepositoryFactory.GetRepository();
```

to

```
IPersonRepository repository = Container.Resolve<IPersonRepository>();
```

Here we see the call to the static method `Resolve`, and we use the interface as the generic type. (Note: This calling syntax is similar to the syntax used by Castle Windsor and Unity – two full-featured IoC Containers.)

Since the `Resolve` method is using the generic type to get the configuration settings, we will need to update the configuration as well. Our "key" needs to be updated to the fully-qualified type name:

```
<add key="PersonRepository.Interface.IPersonRepository"
     value="PersonRepository.CSV.CSVRepository, PersonRepository.CSV,
            Version=1.0.0.0, Culture=neutral"/>
```

Now, our IoC container is able to resolve any number of different types; we just need to add the appropriate configuration entries.

Not a Full IoC Container

Again, this is a "low budget" Inversion of Control container. Some features of full IoC containers include object lifetime management and discovery.

Object lifetime management has to do with how container-requested objects are managed. For example, the container can be configured to return a new instance each time (which is what our sample does), or it can be configured to instantiate an object once, keep track of it, and then return that instance with subsequent requests. The container would also manage when the instances are destroyed.

Discovery is the ability for the container to automatically figure out what objects are available without specific configuration for each type. As one example, the container could scan the assemblies in a specified folder (this could be the application folder or a separate folder designed to hold dynamically-loaded objects). Here's how that would work in our sample: first, we no longer have the configuration entry with the fully-qualified type name. Instead, the container would scan the assemblies and catalog the types (such as cracking open the Repository.CSV.dll and finding the `CSVRepository` type that implements `IPersonRepository`). Then, when the application requests an `IPersonRepository`, it would return an instance of the type that it automatically discovered. Note: this assumes that we only have a single assembly in the folder with types that implement `IPersonRepository`. If we had multiple types, then we would need additional information to decide which type to return.

Does this sound like a lot of work? It is. This is why we usually reach for a fully-implemented container (built by someone else) when we are interested in using the Inversion of Control pattern.

Dependency Injection

Now that we've seen how we can do late-binding with our low-budget IoC container, I need to mention that there is much a better approach for complex applications: using Dependency Injection. We've intentionally kept the examples here simple so that we can focus on the benefits we get out of using Interfaces. But there are several DI patterns (such as Constructor Injection and Property Injection) that allow you to configure a DI container at application start up and then have all of the dependencies (such as our repository) automatically injected into the classes that use them. This is a cleaner way of separating the responsibilities of each class.

Dependency Injection is a much bigger topic (but the topic becomes much easier to understand once you have a good understanding of Interfaces and Abstract Classes). If you want to delve further into Dependency Injection, I would highly recommend picking up a copy of *Dependency Injection in .NET* by Mark Seemann. It is a great resource for learning how to properly use Dependency Injection and also provides a good overview of various IoC/DI containers that are available.

Unit Testing

We have decoupled our application from the repository implementation. This has the effect of adding a "seam" to our code where we can easily swap repository implementations in and out. This is great for unit testing. With our current design, we can easily create a "fake" repository that returns hard-coded data. This way, we could have a test that targets a specific function of our application without worrying about potential problems in the repository (such as a database being off line or a file not being available). These items can be isolated and tested separately. Since they are decoupled, it is less likely that problems with one area will affect the others. The downloadable code contains a "PersonRepository.Fake" project that you can add to the solution and to experiment with.

As mentioned earlier, we can also unit test our repository implementations to make sure that they behave as expected. This is especially important when using late-binding since the errors only show up at runtime. By having comprehensive tests in place, we reduce the likelihood of runtime errors.

Deciding Between an Interface and an Abstract Class

Now that we've gone through several samples, we have a pretty good idea how interfaces work and how we can use them. But everything we did with our samples could also be done with an abstract class. So how do we decide which one to use? There are no hard and fast rules, but here are a few tips to help decide.

Shared Implementation

One way to help decide is to ask how much code (if any) will be shared between the implementing classes. In our repository example, there is no code shared between the implementations. In that case, we can lean towards using an interface. If we have methods with common implementation, then we

might want to consider an abstract class. That way, we can put the shared implementation into the common class rather than having to copy and paste code into each implementation.

Frameworks

Often, if we want to use a third-party framework with our application (such as CSLA for business objects), we need to descend application classes from the classes in the framework. In this type of situation, we want to consider using interfaces for our code. If we were to use abstract classes, we may run into problems since each class can only descend from a single base class. We would have to decide whether to descend from our abstract class or the framework class.

.NET Framework Examples

Abstract classes and interfaces are both used extensively in the .NET framework. For example, the MembershipProvider and RoleProvider (part of the ASP.NET security model) are abstract classes. Much of the shared implementation that hooks into the ASP.NET framework is included in the base class. But a number of methods and properties, such as GetAllUsers, are abstract and must be implemented in the descendant classes.

On the other hand, ADO.NET uses interfaces extensively, such as the IDbConnection interface that represents an open connection to a datasource. The datasource could be Microsoft SQL Server, Oracle, Sybase, ODBC, XML, or any other datasource. As you can imagine, there is little (if any) code that can be shared when opening a connection with any of these sources.

General Advice

Because of the advantages and disadvantages of each, a general recommendation is to tend toward using interfaces. This helps mitigate issues that may arise due to the single-inheritance model. Then later we can determine if an abstract class will make more sense (such as if we have shared code across implementations).

More Places to Explore

The repository pattern is only one example of how interfaces can help us decouple our code. This example shows how we can open up our application to extension by other developers. All a developer needs to do is to create a class that implements our interface, and he can extend the application to use any data storage imaginable. By using interfaces to handle our repository, we've been able to decouple our main application from the data store. The application itself does not care where the data comes from or how it gets fetched or saved. This has decoupled our code and made it more extensible and testable.

Many other design patterns are conducive to interface implementation. This is only a starting point; there are plenty of avenues to explore from here.

Happy coding!