Dependency Injection with Spring.Net

* **January 22, 2008**
* **By**[**David Consdorf**](http://www.developer.com/author/David-Consdorf-88990.htm)

**Introduction**

Spring.Net is a framework aimed at providing comprehensive infrastructural support for .Net enterprise applications. At the core of Spring.Net is the concept of Dependency Injection, otherwise known as Inversion of Control. With the example code provided, you can follow along as you take a look at Spring.Net and how it implements this powerful concept.

**Spring.Net: Overview**

Spring.Net is an open source framework that provides a toolset of components that help developers implement industry best practices such as Dependency Injection and Aspect-Oriented Programming. The concept of the Spring.Net framework was adopted from the Java version of the Spring Framework. To date, the Java version of Spring has been used very successfully in thousands of enterprise applications worldwide. With the success of the Java version, it was only a matter of time before Spring.Net came along, and now that it is maturing, Spring.Net is gaining a following in the .Net world.

Here is a quick list of some of the functionality Spring.Net includes:

* Dependency Injection
* Aspect-Oriented Programming
* Data access support including support for NHibenrate (ORM framework)
* Web Framework
* Integration Testing
* Validation Framework
* Dynamic Reflection
* Threading and Concurrency Support
* Object Pooling
* Expression Evaluation
* Services
* More...

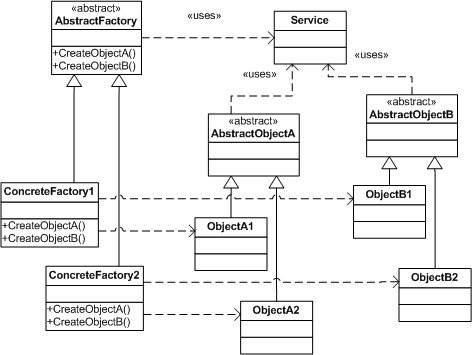
And with the community behind it, Spring.Net is constantly growing with new functionality being added on a regular basis.

Also note that Spring.Net is set up like a toolbox. You are not required to utilize every tool in the box. You can mix and match different functionalities and use only what makes sense for your application. Spring.Net is even split up into different assemblies so you don't need to include the entire framework in your application.

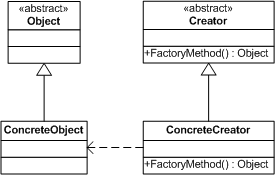
### Dependency Injection with a Factory Pattern

At the core of Spring.Net is the concept of Dependency Injection (DI), also known as Inversion of Control (IoC). The basic idea of Dependency Injection is to provide external dependencies to objects through some external mechanism as opposed to requiring the objects themselves to set up their own dependencies or even worse, requiring the processes that use the objects to set up the required dependencies. The end goal is Loose Coupling. Groups of objects should work independently of one another, focusing on their specific tasks without needing to worry about how other groups of objects work or are constructed.

To better understand the concept of Dependency Injection, it is probably best to look at the classic implementation of DI by using factory classes. As described in the ever useful book, *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, there are two main factory patterns on which factories are built: the Abstract Factory Pattern and the Factory Method Pattern. The idea of the Abstract Factory Pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes, and the idea of the Factory Method Pattern is to provide an interface for creating objects but letting the subclasses decide which classes to implement.



**Figure 1**: Abstract Factory Pattern



**Figure 2**: Factory Method Pattern

To put it simply, factories are a separate object or set of objects that handle the logic for selecting the correct concrete object for a given scenario and constructing the object by injecting any dependencies needed and setting it to a predefined state. Generally, the process using the object will not even know the concrete implementation of the class and instead will work with the object through an interface. By encapsulating the instantiation and construction logic in the factory, the process using the object doesn't need to know specifically what concrete object is being used, how the object is built, or how the object does its job. It only knows that the object will do its job when asked.

In larger applications with multiple possible configurations, factories also can serve as the central point of configuration. For example, if you need to swap out several database components that communicate with MySQL and replace them with a new set of components that communicate with Oracle, all you need to do is change the factory classes to point to the new Oracle components. The rest of the application will be unaffected, because as far as it is concerned, all database components that implement the same interface to retrieve, save, update, and delete objects should work the same. How the components work under the covers and what database they communicate with doesn't really matter.

It is this simple yet powerful idea of a separation of concerns that is the crux of Dependency Injection and Loose Coupling.

Dependency Injection with Spring.Net

* **January 22, 2008**
* **By**[**David Consdorf**](http://www.developer.com/author/David-Consdorf-88990.htm)

**Dependency Injection with Spring.Net**

So, where does Spring.Net fit in? Well, the core of the Spring.Net framework is built on top of the idea of factories and Dependency Injection. Spring.Net is basically just one big configurable object factory. Instead of having to hard-code the often complex factory logic, you simply can provide Spring.Net an XML configuration file to "wire" your objects together.

I could go on, but I have already talked enough about theory. Instead, look at some examples of Spring.Net in action. For this article, I will introduce you to the framework and show you a small sample web application that touches on several of the key Dependency Injection principles of Spring.Net. Note that there are many options for Dependency Injection in Spring.Net and this sample is far from a complete example of everything Spring.Net can do, but it is a good start.

To download the example code in this article, click here: [Download Code Examples](http://www.developer.com/img/2008/01/SpringDIExample.zip).

### Configuring Spring.Net

Start by adding Spring.Net to your project. First, download the spring.Net library at[www.springframework.net](http://www.springframework.net/) and add the assemblies to you project. You should only need the core assembly for this example but the others can be added too.

Next, add the following code to your Web.config or App.config file:

<configSections>

<sectionGroup name="spring">

<section name="context"

type="Spring.Context.Support.ContextHandler,

Spring.Core" />

<section name="objects"

type="Spring.Context.Support.DefaultSectionHandler,

Spring.Core" />

</sectionGroup>

</configSections>

<spring>

<context>

<resource uri="config://spring/objects" />

</context>

<objects xmlns="http://www.springframework.net">

<!—- ToDo: Put mappings here. -->

</objects>

</spring>

**Listing 1**: Spring.Net Configuration

This is just the basic setup configuration for Spring.Net. There isn't much to it really. At this point, you can start adding objects definitions inside the objects tag.

### A Simple Message Object

Start with a simple message object. This example will show you how to create an object definition where the application retrieves an object with a simple string injected into it by Spring.Net.

First, create an object definition and interface as such:

public interface ISimpleMessage {

String Message { get; }

}

public class SimpleMessage : ISimpleMessage {

private String \_message;

public SimpleMessage() {

}

public SimpleMessage(String message) {

\_message = message;

}

public String Message {

get {

return \_message;

}

set {

\_message = value;

}

}

}

**Listing 2**: Simple Message Object

Notice that the interface only provides a method for reading the message and not changing it. This is because the message is expected to be set already when any object reads the message though the interface.

Next, make a mapping for the object in your Spring.Net configuration:

<objects xmlns="http://www.springframework.net">

<!-- Messages -->

<object name="SimpleMessage"

type="SimpleMessage, \_\_Code"

singleton="false">

<constructor-arg name="message"

value="Hello, I am a simple message set by using

constructor-based dependency injection." />

</object>

</objects>

**Listing 3**: Simple Message Object Mapping

What you did here is make a definition for constructing this object. The name of the object is the handle that will be used to retrieve this object. The class simply defines the class of the object instance. (Note that "\_\_Code" is just the generic namespace for web apps. In non-web apps and when referencing assemblies of objects, use the full namespace instead.) Also, note the "singleton" property is set to false in this example. You will learn what this means and more about the singleton pattern later in this article.

Finally, to set the message property of the object, you set a message value as a constructor argument for the object. So now, when the object is initialized, the message will be initialized in the constructor.

Now, run some code to retrieve the object from the Spring.Net context:

// Get Spring.Net context

using (IApplicationContext ctx = ContextRegistry.GetContext()) {

// Get a new instance of the message object

ISimpleMessage simpleMessage1 =

(ISimpleMessage)ctx.GetObject("SimpleMessage");

// Read the message objects message as initialized by Spring.Net

ltlSimpleMessage1.Text = simpleMessage1.Message;

}

**Listing 4**: Retrieve Simple Message from Spring.Net Context

It is pretty easy, right? All this code does is retrieve the Spring.Net application context and then ask the context for an instance of the "SimpleMessage" object. This piece of code only knows the name of the object and what interface to use. It does not know what concrete class is implemented by the interface and it does not have to worry about how to inject the correct message into the object because Spring.Net handles it.

Dependency Injection with Spring.Net

* **January 22, 2008**
* **By**[**David Consdorf**](http://www.developer.com/author/David-Consdorf-88990.htm)

**Constructer Injection vs. Property Injection**

In the last example, you used constructor injection to set the message property. Another way to set the message property is by property injection. Take the last example and add another mapping:

<object name="SimpleMessageByProperty"

type="SimpleMessage, \_\_Code"

singleton="false">

<property name="Message" value="Hello, I am a simple message

set by using property-based dependency injection." />

</object>

**Listing 5**: Simple Message by Property Object Mapping

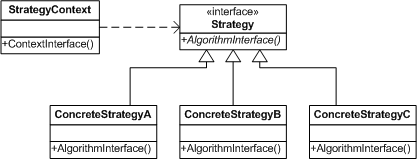
In this example, the no-argument constructor is used to instantiate the object and the message property is set directly. The only difference in the code that calls this object is to ask for a "SimpleMessageByProperty" object instead of a "SimpleMessage" object. Otherwise, the object behaves the same, aside from a slightly different message.

Generally, I like to use constructor injection because it makes it a bit clearer how the object is intended to be initialized. But, there are some cases that require the flexibility of property injection over constructor injection. In the end, it really just comes down to a matter of preference for which type of injection to use.

### Reference Injection Example with the Strategy Pattern

The previous simple message example is a good introduction to mapping an object, but it really doesn't serve as a good example of how Spring.Net can be a powerful tool. Setting a simple string property is nothing special. So now, look at a more interesting example that injects some object references into an object and you'll combine that with the strategy pattern to give it some pizzazz.

First off, for those of you unfamiliar with the strategy pattern, it is a pattern in which you abstract out a specific behavior of an object that may have several different implementations into a separate set of strategy classes. Generally, this is implemented by giving an object a reference to one of several concrete strategy objects though a strategy interface. The strategy interface then is used to run the specified strategy for the desired behavior.



**Figure 3**: Strategy Pattern

The strategy pattern is useful for abstracting out complex calculations, algorithms, or specialized actions that may change sometime in the future. For this example, look at a simple number array class and give it a sort function that utilizes the different possible sorting functions (Bubble Sort, Heap Sort, Quick Sort, and so forth) through the use of strategies.

Here is the code:

public interface INumberArray {

void addNumber(int number);

int[] getNumbers();

void sortNumbers();

}

public class NumberArray : INumberArray {

private int \_numberOfElements = 0;

private int[] \_numbers = new int[25];

private ISortStrategy \_sortStrategy;

public NumberArray(ISortStrategy sortStrategy) {

\_sortStrategy = sortStrategy;

}

public void addNumber(int number) {

\_numbers[\_numberOfElements] = number;

\_numberOfElements++;

}

public int[] getNumbers() {

int[] result = new int[\_numberOfElements];

for (int x = 0; x < \_numberOfElements; x++ ) {

result[x] = \_numbers[x];

}

return result;

}

public void sortNumbers() {

\_sortStrategy.sort(\_numbers, \_numberOfElements);

}

}

public interface ISortStrategy {

void sort(int[] numbers, int numOfElements);

}

public class BubbleSortStrategy : ISortStrategy {

public void sort(int[] numbers, int numOfElements) {

// Bubble Sort Algorithm (See example code)

}

}

public class HeapSortStrategy : ISortStrategy {

public void sort(int[] numbers, int numOfElements) {

// Heap Sort Algorithm (See example code)

}

}

public class QuickSortStrategy : ISortStrategy {

public void sort(int[] numbers, int numOfElements) {

// Quick Sort Algorithm (See example code)

}

}

**Listing 6**: Number Array & Strategies

Dependency Injection with Spring.Net

* **January 22, 2008**
* **By**[**David Consdorf**](http://www.developer.com/author/David-Consdorf-88990.htm)

Notice that the NumberArray class holds a reference to one of the sort strategies as defined by the ISortStrategy interface. The NumberArray class doesn't have any direct reference to a concrete implementation of the strategy. So, how does the strategy get set up, then? Though Spring.Net Dependency Injection, of course! Take a look at the Spring.Net mapping:

<!-- Strategies -->

<object name="BubbleSortStrategy"

type="BubbleSortStrategy, \_\_Code"

singleton="true">

</object>

<object name="HeapSortStrategy"

type="HeapSortStrategy, \_\_Code"

singleton="true">

</object>

<object name="QuickSortStrategy"

type="QuickSortStrategy, \_\_Code"

singleton="true">

</object>

<!-- Number Arrays -->

<object name="NumberArray\_BubbleSort"

type="NumberArray, \_\_Code"

singleton="false">

<constructor-arg name="sortStrategy" ref="BubbleSortStrategy" />

</object>

<object name="NumberArray\_HeapSort"

type="NumberArray, \_\_Code"

singleton="false">

<constructor-arg name="sortStrategy" ref="HeapSortStrategy" />

</object>

<object name="NumberArray\_QuickSort"

type="NumberArray, \_\_Code"

singleton="false">

<constructor-arg name="sortStrategy" ref="QuickSortStrategy" />

</object>

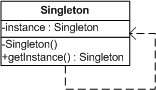
**Listing 7**: Number Array & Strategies Mappings

So, as you can see from the mappings, there are three different ways to initialize the NumberArray, each with a different strategy object injected. The code using a NumberArray object just needs to ask Spring.Net for a "NumberArray\_BubbleSort", "NumberArray\_BubbleSort", or "NumberArray\_BubbleSort" object. Each NumberArray object will look the same to the code using it, but behind the scenes the currently initialized strategy will be used to run the sort functionality.

To see an example of the code above being used in a simple web app, check out the example code for this article: [Download Code Examples](http://www.developer.com/img/2008/01/SpringDIExample.zip).

### Singleton Pattern

Another important concept employed by Spring.Net is the use of the singleton pattern. You will notice in the examples above that there is a "singleton" property for each object mapping, but what does this mean? To understand this further, look at the singleton pattern:



**Figure 4**: Singleton Pattern

Basically, the singleton pattern ensures that there is only one instance of a class and that you can retrieve and use that single instance through a global point of access. Usually, this is done by implementing a class with a private constructor so it can only be instantiated internally. Then, the class instantiates a single instance of the object within itself and provides a public static reference to the instance so the application can reference it. This pattern is very similar to using static functions and properties within a class, but it provides a bit more flexibility and control because you get an actual object instance to work with.

Spring.Net takes full advantage of the singleton pattern by allowing users to specify objects as singleton or not. So, for instance, if you look at the NumberArray example above, the NumberArray objects defined in the Spring.Net configuration are not singletons but the strategy objects defined are singletons. What this means is that each time you ask Spring.Net for a NumberArray object, it will create a new NumberArray object instance. This makes sense for the NumberArrays because you can have several number arrays in different states with different numbers in the array.

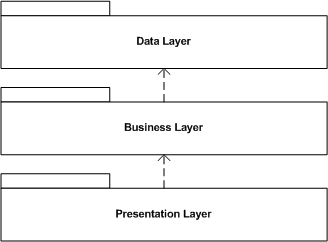
The strategy classes, however, are not really concerned with state. They have no properties that need to be maintained. They are simply used as plug and play functionality, so it makes sense to define them as singletons. When Spring.Net is asked for a strategy object, Spring.Net always returns the same singleton instance. And, when Spring.Net injects the strategies into the NumberArray objects, Spring.Net always injects the same strategy instance. Each strategy object will have only one instance per application.

It is important to note that the singleton pattern is not really necessary in the NumberArray example above. Each NumberArray could use a new instance of a strategy object, and everything would work just as well. The only problem with this case, though, is that your application will use extra time to create all the new objects, use extra memory to store each of the new objects, and then use extra time to clean up the objects when they are no longer being used. Generally, it is a much better practice to use the singleton pattern when it makes sense. It will save your application time and space because only one object will be initialized and it will only happen once.

### Quick N-Tier Example

For the last example, look at a common usage of Spring.Net Dependency Injection where an application is split into multiple tiers and wired together by Spring.Net.

Anyone involved in larger applications has probably encountered the concept of N-tier application architecture. The basic idea is that an application is split up into multiple tiers of functionality. Each tier handles its own concerns and only communicates with the tier above/below it. For example, you might have a 3-tier application architecture as such:



**Figure 5**: 3-Tier Application Architecture Example

Dependency Injection with Spring.Net

* **January 22, 2008**
* **By**[**David Consdorf**](http://www.developer.com/author/David-Consdorf-88990.htm)

In this example, there is a presentation/view layer that takes care of displaying the application data either though a web page, client application, mobile application, or whatever. Then, you have a service/business layer that takes care of the application logic such as any calculations, operations, and so on. And finally, you have a data layer that handles communications with the database, XML, or whatever other data sources the application might have. Each layer communicates only with the other layers via an interface, but each layer has no idea how the other layers are setup or how they operate.

For example, the presentation/view layer will ask the service layer to create a new business object of some sort for the given inputs from the user, the service layer will do some logic to build an object based on the inputs and then it will call the data layer to persist the object. The service layer doesn't need to concern itself with what type of application the presentation layer collected the inputs with (in other words, Web or Client) and the service layer also doesn't care how the data layer persists the object (in other words, MS SQL, DB2, or XML file). This makes the application easily configurable for different uses. If the one layer needs to change, only that layer needs to be rewritten and the rest can be reused.

Spring.Net and Dependency Injection help implement multi-tier application architectures by splitting an application into configurable components. With Spring.Net, it is easy to keep all the components of one layer separate from the other layers and to inject dependencies with other layers transparently through the use of XML configuration files.

Take a quick look at some example code for wiring a Service object from the service/business layer with a data access object (DAO) from the data layer:

public interface IServiceExample {

void doOperationOnObject(Object obj);

Object getObject(int objectID);

void saveObject(Object obj);

}

public class ServiceExample : IServiceExample {

public IDAOExample \_daoExample;

public ServiceExample(IDAOExample daoExample) {

\_daoExample = daoExample;

}

public void doOperationOnObject(Object obj) {

// Enter business logic for operation here

}

public Object getObject(int objectID) {

// Use DAO to retrieve object from the database

return \_daoExample.getObject(objectID);

}

public void saveObject(Object obj) {

// Use DAO to save object to the database

\_daoExample.saveObject(obj);

}

}

public interface IDAOExample {

Object getObject(int objectID);

void saveObject(Object obj);

}

public class DAOExample : IDAOExample {

public DAOExample() {

}

public Object getObject(int objectID) {

// Put database code to retrieve object here

}

public void saveObject(Object obj) {

// Put database code to save object here

}

}

**Listing 7**: Service & DAO

<!-- Service & DAO -->

<object name="ServiceExample"

type="ServiceExample, \_\_Code"

singleton="true">

<constructor-arg name="daoExample" ref="DAOExample" />

</object>

<object name="DAOExample"

type="DAOExample, \_\_Code"

singleton="true">

</object>

**Listing 8**: Service & DAO Mappings

In this simple example, you have a service object that communicates with a DAO object to save objects to the database. The DAO object is injected into the service object via Spring.Net. By using injection, there are no specific links between layers within the code.

Say your application is moving from a MySQL database implementation to an Oracle database implementation. All you would need to do is to create a new DAO class for Oracle, and then switch the Spring.Net configuration to inject the new Oracle DAO object into the service instead of the MySQL DAO object. Nothing in the service layer would need to change at all.

Note that in this case, the presentation layer using the service objects would generally either call Spring.Net to get the service objects or call some sort of service manager to get the service. Spring.Net also provides other options worth looking into like their web framework which allows you to directly inject the service objects into ASP.Net pages.

Also note that both the service and DAO objects are singletons because they are used just for their operations and don't hold any specific state.

Dependency Injection with Spring.Net

* **January 22, 2008**
* **By**[**David Consdorf**](http://www.developer.com/author/David-Consdorf-88990.htm)

**Testing with Dependency Injection**

Another interesting use of Dependency Injection is for unit and integration testing. The ability to swap out different components easily with Spring.Net allows developers to create stub components to test with so they can focus on testing their specific piece of code without needing to worry about other parts of the system being completed and/or working properly.

For example, say you are developing some services for an application's business layer but the data layer is not implemented yet or there are environment restrictions that don't allow you to access the needed data sources. With Spring.Net and Dependency Injection, you still can test the service by creating some mock data components that fill in the missing data layer functionality needed to complete your tests. Then, all you need is a separate Spring.Net configuration using your testing components. Later, when you need to integrate with the real data layer, you can just swap out the testing configuration and put in the integration testing or production configuration.

There are also other testing techniques with Spring.Net including Spring.Net's own testing framework, so if you are interested, check out some other articles focused on the subject or check out the Spring.Net web site: [www.springframework.net](http://www.springframework.net/).

### More Spring.Net

It is important to note again that although this article focuses mainly on the Dependency Injection aspects of Spring.Net, there is much more to Spring.Net than object factories. Spring did start with the idea of being a robust configurable object factory, but that idea lent itself well toward expanding outward into a toolset. The whole idea of object factories, after all, is to provide a mechanism for building applications as set of components, so it makes sense that when Spring.Net expands the result is new, reusable components to help solve common problems or implement new ways to build applications.

With that said, make sure to check out [www.springframework.net](http://www.springframework.net/) and read more about Spring.Net and what tools it can offer to help you build a great application.

### Conclusion

I hope this article helped shed some light on Spring.Net and how it can be an incredible resource for implementing Dependency Injection in your applications. Dependency Injection and Loose Coupling are powerful concepts that will ultimately make your applications more flexible, extendable, and easier to maintain. Be sure to check out Spring.Net for your Dependency Injection needs and also be sure to check out the other wonderful functionality Spring.Net provides.