

# Inversion of Control and Dependency Injection with Castle Windsor Container - Part I

Inversion of Control (IoC) and Dependency Injection (DI) are two related practices in software development which are known to lead to higher testability and maintainability of software products. While some people employ them daily in their work, many others still don't know much about them, mostly because they require in the former a shift in the usual thinking process.

This article introduces these notions in a simple and affordable fashion, with a real example which evolves step by step to take advantage from IoC and DI, leveraging the features offered by an open source framework called [Castle Project](#).

## Inversion of Control/Dependency Injection Series

- [IOC/DI - Part I](#) This article introduces Inversion of Control/Dependency Injection in a simple and affordable fashion, with a real example which evolves step by step to take advantage from IoC and DI, leveraging the features offered by an open source framework called Castle Project.
- [IOC/DI - Part II](#) In the previous article I introduced the concepts of Inversion of Control and Dependency Injection, and showed how to benefit from their use when developing a simple application. In this article I will resume the previous discussion extending the requirements of the former example to show how IoC deals with changes and what is its real potential.
- [IoC/DI - Part III](#) In this article I conclude the discussion about the core features of Windsor Container while evolving the simple example that accompanied you through the whole series of these articles. The next article will tackle more advanced topics.
- [IoC/DI - Part IV](#) In this article I introduce the missing core features I didn't tackle before. They require a little more knowledge of Windsor, which I hope I've been able to hand down so far.

## Introduction

IoC and DI are still pretty obscure topics in .NET software development, mainly because they require a little bit of effort in the beginning to start making sense of their use; and maybe even because they've never been promoted a lot nor implemented by Microsoft until lately with its [ObjectBuilder](#) open source framework. However, their understanding is not a waste of time, because they doubtless lead to better structured, modularized, testable and maintainable applications. Once they've discovered them, developers never get back to their old way of building software.

Furthermore, a lot of open source projects and frameworks supply such functionalities out of the box. In this article I'm going to show how to benefit from IoC and DI using [Castle Project](#). Other implementations are available as well, like those which come along with [Spring.NET](#) and [StructureMap](#). The choice depends mostly on individual preferences and additional needs of the application, which could benefit from other facilities supplied by the framework.

StructureMap is the lightest and only provides an IoC container. Castle Project supplies other functionalities like data persistence, an MVC framework for web applications; and is very actively developed. Finally, Spring.NET is the wider and most mature since ported from the Java [SpringFramework](#); but in most cases you won't need all of its features.

In this article I'm using Castle Project's IoC implementation, since I find its syntax more intuitive and with some interesting additional features not provided by the others. Remember that the choice is mostly a matter of individual tastes.

## Requirements of the sample application

In order to approach IoC and DI concepts I'm presenting a simple example of interaction between objects, first resolving it with a canonical approach and then leveraging IoC and DI to show how much the structure of the application gains in flexibility and all the other characteristics I've listed in the previous paragraph. The requirement is to write an application capable of retrieving the title of an HTML document downloaded from the web. The steps are pretty simple - you need to download the file from the network using the HTTP protocol and then apply some parsing to it in order to extract the text wrapped in the `title` tag.

## About IoC and DI

I'm not delving too much in the formal aspects of these two patterns since I want this article to be an

introduction and a quick start guide to let you grasp the core topics. For an in-depth discussion I redirect you to the references included at the bottom of this page. Instead, I'm going to describe how IoC and DI should be approached by developers who never worked with them.

At a glance, these patterns are said to be based on the [Hollywood Principle](#), which states: "don't call us, we'll call you". With a canonical approach, you hard code the classes of the objects you want to instantiate in the source of your application, supply parameters to their constructors and manage their interactions. Each object knows at compile time which are the real classes of the objects they need to interact with, and they will call them directly. So, under this point of view, you and your objects are the ones calling Hollywood. To invert this approach, you need some support from a framework which makes your application smart enough to guess which objects to instantiate, how to instantiate them and, in general, how to control their behavior. Instead of working with concrete classes you'll work with abstractions like interfaces or abstract classes, letting your application decide which concrete classes to use and how to satisfy their dependencies on other components. This concept may sound a bit weird in the beginning, but you'll see that this makes a lot of sense.

## Creating a simple web page scraper

Following the requirements of the sample application let's write a class capable of satisfying them. It's called `HtmlTitleRetriever`, and exposes a single method called `GetTitle`, which accepts the `Uri` of a file and returns a string with the title of the HTML document -if it has one- or an empty string.

```
01. public class HtmlTitleRetriever
02. {
03.     public string GetTitle(Uri file)
04.     {
05.         string fileContents;
06.         string title = string.Empty;
07.
08.         WebClient client = new WebClient();
09.         fileContents = client.DownloadString(file);
10.
11.         int openingTagIndex = fileContents.IndexOf("<title>");
12.         int closingTagIndex = fileContents.IndexOf("</title>");
13.
14.         if(openingTagIndex != -1 && closingTagIndex != -1)
15.             title = fileContents.Substring(openingTagIndex,
16.                 closingTagIndex - openingTagIndex).Substring(7);
17.
18.         return title;
19.     }
20. }
```

What this class does is very simple. First it instantiates a `WebClient` object - a facade to ease use of `HttpWebRequest` and `HttpWebResponse` classes. Then it uses the object to retrieve the contents of the remote resource, using the HTTP protocol. Using string routines, it looks for the opening and closing `title` tags and extracts the text between them.

At this point you might be wondering what's so wrong in this class to imply the need for a different approach in implementing its requirements. Actually, not much, as long as the requirements remain so simple. But from a more general point of view there are at least two aspects which need to be revisited about this implementation:

- The class does more than it should do. A principle of good system design is [SoC](#) - separation of concerns. According to this principle, a software component should be able to do a simple task only, and do it well. Instead, the class first downloads the file from the web, and then applies some sort of parsing to retrieve the contents it cares about. These are two different tasks, which should be separated into two different components.
- What if the class needed to be able to retrieve documents not accessible via the HTTP protocol? You'd need to change the implementation of the class to replace or add this feature. The same consideration applies to the parsing process. In this example it doesn't make much sense but you may discover that under certain circumstances adopting a different scraping mechanism would lead to better performance. In other words, the class has deep knowledge - read, dependencies - on concrete implementations of other components. It's better to avoid this because it leads to bad application design.

## Components and Services

Since I'm going to implement the solution using Castle Project, I will adopt the same terminology used

in its documentation. I'm reporting the exact definition given by [Hamilton Verissimo](#) - Castle's administrator - in its introductory article about Castle (referenced at the bottom of the page):

*"A component is a small unit of reusable code. It should implement and expose just one service, and do it well. In practical terms, a component is a class that implements a service (interface). The interface is the contract of the service, which creates an abstraction layer so you can replace the service implementation without effort."*

Components and services are the basic concepts you'll need to use when working with Castle. So, with this in mind, I'm going to make a little step towards a better application design.

## Applying SoC with Components and Services

So far you've seen that the responsibilities of the `HtmlTitleRetriever` class can -and should- be separated into two classes: One for retrieving files and one for parsing their contents.

Note that these are generic jobs, in that they can be implemented in several ways. The implementation above is just one of the available choices, but you can think of retrieving files from other mediums, as well as adopt a different mechanism to extract the contents of the title tag. In other words, these tasks are supplied by a service, which can be carried out in several ways. The concrete classes which perform the task are the components. The file downloading and title scraping service' contracts can be defined via interfaces, `IFileDownloader` and `ITitleScrapper`.

```
01. public interface IFileDownloader
02. {
03.     string Download(Uri file);
04. }
05.
06. public interface ITitleScrapper
07. {
08.     string Scrape(string fileContents);
09. }
```

Now let's implement these services with concrete classes - the components - supplying the same features as the original `HtmlTitleRetriever` class.

```
01. public class HttpFileDownloader : IFileDownloader
02. {
03.     public string Download(Uri file)
04.     {
05.         return new WebClient().DownloadString(file);
06.     }
07. }
08.
09. public class StringParsingTitleScrapper : ITitleScrapper
10. {
11.     public string Scrape(string fileContents)
12.     {
13.         string title = string.Empty;
14.         int openingTagIndex = fileContents.IndexOf("<title>");
15.         int closingTagIndex = fileContents.IndexOf("</title>");
16.
17.         if(openingTagIndex != -1 && closingTagIndex != -1)
18.             title = fileContents.Substring(openingTagIndex,
19.                 closingTagIndex - openingTagIndex).Substring(7);
20.
21.         return title;
22.     }
23. }
```

These components completely satisfy the requirements of the application. Now they need to be assembled to provide the downloading and parsing services together. So let's modify the original awful class to benefit from their features. This time the class mustn't be aware of the concrete implementation of the services. It just needs to know that someone will provide those services and it will simply use them. The new `HtmlTitleRetriever` class now looks like this:

```
01. public class HtmlTitleRetriever
02. {
03.     private readonly IFileDownloader dowloader;
04.     private readonly ITitleScrapper scrapper;
05. }
```

```
06.     public HtmlTitleRetriever(IFileDownloader downloader,
07.         ITitleScraper scraper)
08.     {
09.         this.downloader = downloader;
10.         this.scraper = scraper;
11.     }
12.
13.     public string GetTitle(Uri file)
14.     {
15.         string fileContents = downloader.Download(file);
16.         return scraper.Scrape(fileContents);
17.     }
18. }
```

You can see that it now provides a parameterized constructor which accepts two references to components supplying the two services defined above. Again, note that it doesn't have any knowledge of how those services carry out their work. It only knows that they do, and hopefully they do it well.

Now it looks like the former problem has been solved, but instantiating the `HtmlTitleRetriever` class is trickier than before, since there are dependencies on the constructor to be satisfied. Here's what you'd need to do to create an instance:

```
1. IFileDownloader downloader = new HttpFileDownloader();
2. ITitleScraper scraper = new StringParsingTitleScraper();
3. HtmlTitleRetriever retriever = new HtmlTitleRetriever(downloader, scraper);
```

That's more work than you would like to do, of course, and for a moment imagine a class which exposes a service which needs many more dependencies to be satisfied, which, in turn, have dependencies on other services. It would become a nightmare very fast. Here's where IoC and DI come to rescue. Believe it or not, you will be able to instantiate the `HtmlTitleRetriever` class without supplying its constructor with any of the dependencies it needs. Actually, you won't even need to call its constructor. Someone else will do it on your behalf.

## Approaching IoC and DI

Managing objects creation and disposal using IoC and DI requires actually less magic than I pretended to make you believe. So, who is going to deal with objects if you are no longer in charge for it?

The main point of a framework which offers IoC and DI is a software component called *container*. As its name implies, the container will achieve knowledge about components needed by your application to run and will try to be smart enough to understand which component you want. This happens when you query it asking to return an instance of one of the components it contains. This is what IoC means in practice; you'll no longer instantiate classes using constructors, but instead register them into the container and then ask it to give you one instance of a component. This can be done in several ways and with a lot of configuration options which you'll see later and in the next article.

The other fundamental feature of the container is that it will be able to resolve-and inject-dependencies between your objects; hence the name *Dependency Injection*. In the sample application, the container will be smart enough to guess that in order to instantiate an `HtmlTitleRetriever` object, it needs to instantiate components supplying the `IFileDownloader` and `ITitleScraper` services.

## Castle Microkernel

Castle Project IoC container is offered in two flavors. The `MicroKernel`, a lightweight container which offers the core functionalities of IoC and DI. `Windsor Container`, built on top of the `MicroKernel` and extending its features by adding support for external configuration and interceptors. Most of the times you will end up using `Windsor`, but to keep things simple I'll start by illustrating how to achieve IoC and DI with the `MicroKernel`.

Below is the code needed to configure an application to make use of the `HtmlTitleRetriever` class via the `MicroKernel`.

```
01. IKernel kernel = new DefaultKernel();
02.
03. kernel.AddComponent("HttpFileDownloader", typeof(IFileDownloader),
04.     typeof(HttpFileDownloader));
05. kernel.AddComponent("StringParsingTitleScraper", typeof(ITitleScraper),
06.     typeof(StringParsingTitleScraper));
07. kernel.AddComponent("HtmlTitleRetriever", typeof(HtmlTitleRetriever));
```

```

07. HtmlTitleRetriever retriever = (HtmlTitleRetriever)
    kernel[typeof(HtmlTitleRetriever)];
08.
09. // do some work..
10.
11. kernel.ReleaseComponent(retriever);

```

The steps involved in the configuration and utilization of the `MicroKernel` API are described below:

- A new instance of the container is created.
- One by one all the components are registered into the container using the `AddComponent` method, which exposes several overloads. The first parameter is a key used to identify the component. Then you can either specify both the contract and concrete class of the component or just the implementation, in case the component doesn't implement any interface.
- An instance of the `HtmlTitleRetriever` class is obtained from the container specifying the type. You could use the key as well. In this phase, the container sees that in order to instantiate the class it needs to supply two parameters to its constructor. Noticing that the parameters are of type `IFileDownloader` and `ITitleScraper` it realizes that two components exposing such services have already been registered, and simply instantiates and supplies them to the constructor of the `HtmlTitleRetriever` class.
- After using the instance of the class you should release it by calling the `ReleaseComponent` method on the container. This is not mandatory, since the container will handle it automatically in most common situations, but there are cases where you need more granularity over components lifecycle and want to decide when to release them.

## Castle Windsor Container

Even though the `MicroKernel` provides enough features for this simple example, `Windsor Container` is usually more suitable for applications which require a more flexible approach to container configuration and a more user friendly API. The snippet below shows how to configure the sample application with `Windsor Container`.

```

01. IWindsorContainer container = new WindsorContainer();
02.
03. container.AddComponent("HttpFileDownloader", typeof(IFileDownloader),
04.     typeof(HttpFileDownloader));
05. container.AddComponent("StringParsingTitleScraper", typeof(ITitleScraper),
06.     typeof(StringParsingTitleScraper));
07. container.AddComponent("HtmlTitleRetriever", typeof(HtmlTitleRetriever));
08.
09. HtmlTitleRetriever retriever = container.Resolve<HtmlTitleRetriever>();
10.
11. string title = retriever.GetTitle(new Uri("some uri..."));
12.
13. container.Release(retriever);

```

As you can see, the API is very similar. That's because `Windsor` is not another container, but it's built on top of the `MicroKernel` and simply augments its features. One small but useful feature to note is that `Windsor` lets you retrieve components using generics, thus avoiding casts.

So far, you've seen how to configure the container programmatically. In a real life application, you would need to write a lot of code for container configuration. Changing something would require a new build of the entire solution. `Windsor` offers a new feature which lets you configure the container using XML configuration files, much like you would do with a standard `.NET` application. So let's rewrite the code above to benefit from external configuration.

First you need to create a configuration file, called `App.config` or `Web.Config`, depending on the kind of application you're building. Note that `Windsor` has the ability to read configuration from other locations as well. The default application configuration file is just one of the options.

```

01. <?xml version="1.0" encoding="utf-8" ?>
02. <configuration>
03.
04.     <configSections>
05.         <section name="castle"
06.             type="Castle.Windsor.Configuration.AppDomain.CastleSectionHandler,
07.             Castle.Windsor" />
08.     </configSections>
09.

```

```
11.     <castle>
12.       <components>
13.         <component id="HtmlTitleRetriever"
14.           type="WindsorSample.HtmlTitleRetriever, WindsorSample">
15.       </component>
16.       <component id="StringParsingTitleScrapper"
17.         service="WindsorSample.ITitleScrapper, WindsorSample"
18.         type="WindsorSample.StringParsingTitleScrapper,
19.           WindsorSample">
20.       </component>
21.       <component id="HttpFileDownloader"
22.         service="WindsorSample.IFileDownloader, WindsorSample"
23.         type="WindsorSample.HttpFileDownloader, WindsorSample">
24.     </components>
25.   </castle>
26.
27. </configuration>
```

First, the section handler for Windsor configuration has to be registered into the `configSections` section. Then, the actual configuration takes place into the `castle` section and is very similar to what I did before via code. The syntax is the following:

- `id` (required): a string name to identify the component
- `service`: the contract implemented by the component
- `type`: the concrete type of the component

`Service` and `type` attributes require the full qualified name of the type (`namespace.typename`) and the assembly name after the comma.

You may have noticed that the `HtmlTitleRetriever` class is registered without supplying a service. In fact, it doesn't implement any interfaces nor base class, since it's unlikely that you will ever provide a different implementation of it. The other two components, instead, are concrete implementation of a service that can be carried out in several ways. This syntax lets you register more than one component for the same service. By default, when the container finds that two components have been registered for the same service, it resolves dependencies by supplying the first component registered—either in the configuration file or via code—but this behavior can be changed using the string identification key of the component. Here's how the code of the application changes to take advantage of external configuration:

```
1. IWindsorContainer container = new WindsorContainer(new XmlInterpreter());
2.
3. HtmlTitleRetriever retriever = container.Resolve<HtmlTitleRetriever>();
4.
5. string title = retriever.GetTitle(new Uri("some address..."));
6.
7. container.Release(retriever);
```

The main difference is that the constructor of the container is called with an instance of the `XmlInterpreter` class, which by default reads the configuration from the application configuration file.

## Taking advantage of IoC and DI

So far you've seen how to switch from a canonical programming process to inversion of control. Now let's make a small step towards understanding why IoC makes applications better.

Suppose that the original requirements changed and you needed to retrieve files no longer using the HTTP protocol but instead via FTP. With the former approach you'd need to change the code into the `HtmlTitleRetriever` class. That's not a lot of work since the example is very simple, but in an enterprise application this may imply a lot of work. Instead, let's see what it takes to provide this feature using Windsor.

First, you'll need to create a class implementing the `IFileDownloader` interface which retrieves files via FTP. Then, register it into the configuration file, replacing the former HTTP implementation. So, no need to change a single line of code of the application, and no need for a recompilation since you can provide this new class into a new assembly. Actually, the features provided by Windsor are much smarter than this, but this is topic for another article.

## Summary

In this article you've seen what Inversion of Control and Dependency Injection are and how they can lead to a better design of a software application. You've seen how to take advantage of them using the open source Windsor Container which comes along with Castle Project. In the next article I will delve deeper into the features offered by the container and how they can make our applications a pleasure to write, test and maintain.

## References

- Martin Fowler - [Inversion of Control Containers and the Dependency Injection pattern](#) - 2004
- Hamilton Verissimo - [Introducing Castle, Part I](#) - 2004
- Oren Eini - [Inversion of Control and Dependency Injection: Working with Windsor Container](#) - 2006
- Alex Henderson - [Container Tutorials](#) - 2007
- [Castle Windsor Container documentation](#)

Original Url: <http://dotnetslackers.com/articles/designpatterns/InversionOfControlAndDependencyInjectionWithCastleWindsorContainerPart1.aspx>