Interface 설계 – Part 3

Part 2에서는 Entity 즉 객체의 속성 위주로, 무엇을 표현할 수 있는지에 대해서 추상화해서 Interface를 제작하는 방법을 설명했습니다.

이번에는 Service 즉 행동(Action)에 해당하는 Method들을 추상화해서 Interface를 제작하는 방법을 설명하겠습니다.

Interface 명명 방법에서도 언급했지만 –er / -or 로 끝나는 Interface는 대부분 Method에 대한 추상화를 수행한 것들이 대부분일 것입니다.

이러한 예를 가지고, 설명해 보겠습니다. 아래 코드의 ICompressor 는 압축/복원을 수행한다는 Interface입니다.

/// <summary>

/// 압축/복원을 수행하는 Class의 기본 Interface

/// </summary>

public interface ICompressor

{

/// <summary>

/// 지정된 데이타를 압축한다.

/// </summary>

/// <param name="input">압축할 Data</param>

/// <returns>압축된 Data</returns>

byte[] Compress(byte[] input);

/// <summary>

/// 압축된 데이타를 복원한다.

/// </summary>

/// <param name="input">복원할 Data</param>

/// <returns>복원된 Data</returns>

byte[] Decompress(byte[] input);

}

아시다시피, 압축 알고리즘은 다양합니다. GZip, Deflate, BZip2, 7Zip, Cab 등등… 여기서는 압축 알고리즘에 대한 설명은 생략하기로 하고, 다양한 압축 알고리즘을 어떻게 추상화 했는지 파악해 보겠습니다. 알집 등 압축 Utility를 보면, 다중파일 압축, 압축된 파일을 여러 개로 분리 관리하는 기능 등 상당히 많지만, 압축기능 자체는 정보를 압축/복원하는 기능이다. 이에 많은 수형을 압축할 수 있겠지만, 가장 기본이 되는 byte array만 제공한다. 물론 수형에 따라서 IByteCompressor라고 붙일 수도 있지만, 압축대상이 되는 Data가 일반적으로 byte array이므로 대상 수형을 생략했다.

수형이나 Data를 중점을 두고 클래스를 만드는 경우는 .NET의 IFormatter 를 상속받는 BinaryFormatter, SoapFormatter 등을 들 수 있다.

어쨌든 위와 같은 ICompressor를 정의하였다면, 실제 구현은 각각의 Algorithm별로 따로 구현해야 한다. 물론 공통 부분이 있다면, Abstract class (.NET에서는 Base class)를 정의하고, 각각의 concrete class에서 구현해야할 메소드는 abstract로 남겨두면 된다. 하지만 이 경우에서는 공통 구현 부분이 없으므로, 각각의 concrete class를 따로 구성합니다.

실제로 GZipCompressor, DeflateCompressor, 7ZipCompressor 등을 제작하였다 (물론 핵심 압축 알고리즘은 .NET Framework에 있거나, Open Source의 알고리즘을 이용하였다)

GZipCompressor 예

/// <summary>

/// GZip Algorithm을 이용하여 압축/복원을 수행하는 Class

/// </summary>

public class GZipCompressor : ICompressor

{

#region << log4net >>

private static readonly log4net.ILog log =

log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);

#endregion

#region << Implementation of ICompressor >>

/// <summary>

/// 지정된 데이타를 압축한다.

/// </summary>

/// <param name="input">압축할 Data</param>

/// <returns>압축된 Data</returns>

public virtual byte[] Compress(byte[] input)

{

if (log.IsDebugEnabled)

log.Debug(CompressorUtil.SR.CompressStartMsg);

// check input data

if (input.IsZeroLength())

{

if (log.IsWarnEnabled)

log.Warn(CompressorUtil.SR.InvalidInputDataMsg);

return new byte[0];

}

byte[] output = null;

using (MemoryStream outStream = new MemoryStream())

{

using (GZipStream gzip = new GZipStream(outStream, CompressionMode.Compress))

{

gzip.Write(input, 0, input.Length);

}

output = outStream.ToArray();

}

if (log.IsDebugEnabled)

{

log.DebugFormat(CompressorUtil.SR.CompressResultMsg,

input.Length, output.Length, output.Length/(double) input.Length);

}

return output ?? new byte[0];

}

/// <summary>

/// 압축된 데이타를 복원한다.

/// </summary>

/// <param name="input">복원할 Data</param>

/// <returns>복원된 Data</returns>

public virtual byte[] Decompress(byte[] input)

{

if (log.IsDebugEnabled)

log.Debug(CompressorUtil.SR.DecompressStartMsg);

// check input data

if (input.IsZeroLength())

{

if (log.IsWarnEnabled)

log.Warn(CompressorUtil.SR.InvalidInputDataMsg);

return new byte[0];

}

byte[] output = null;

MemoryStream outStream = new MemoryStream();

try

{

using (MemoryStream inStream = new MemoryStream(input))

using (GZipStream gzip = new GZipStream(inStream, CompressionMode.Decompress, true))

{

int readCount;

byte[] buffer = new byte[CompressorUtil.BUFFER\_SIZE];

while ((readCount = gzip.Read(buffer, 0, CompressorUtil.BUFFER\_SIZE)) > 0)

{

outStream.Write(buffer, 0, readCount);

}

output = outStream.ToArray();

}

}

catch (Exception ex)

{

if (log.IsErrorEnabled)

log.Error(ex);

throw;

}

finally

{

outStream.Close();

}

if (log.IsDebugEnabled)

{

log.DebugFormat(CompressorUtil.SR.DecompressResultMsg,

input.Length, output.Length, output.Length/(double) input.Length);

}

return output ?? new byte[0];

}

#endregion

}

뭐 특별날 건 없고, System.IO.Compression.GZipStream 을 사용하여 ICompressor를 구현한 것이라 볼 수 있습니다.

자 이제… Interface도 정의하였고, 몇 개의 Concrete Class도 구현하였습니다. 이렇게 만든 이유가 뭐죠?

흠… 사실 IoC (Inversion Of Control) / DI (Dependency Injection)을 모른다 하더라도 Factory Pattern 정도를 알고 있다면, 더 자세한 설명과 의의를 설명할 수 있겠지만…

우선은, 외부에 어떤 Algorithm을 이용해서 압축/복원하는지를 알리지 않고도 (물론 Reflection을 통해 instance의 type을 조사하면 알 수는 있습니다만), ICompressor를 사용하는 모든 사용자에게 똑 같은 메소드를 제공한다고 볼 수 있습니다. (즉 사용자와 제공자가 약속만 있고, 실제 수행여부에 대해서는 사용자에게 알려주지 않는 것이죠)

자 그럼 Factory Pattern이나 IoC 개념을 어느정도 안다고 한다면, 시스템에서 사용할 압축 알고리즘을 코드상에서 정의하는 것이 아니라, 환경설정에서 정의하고, 그 설정값에 따라 Compressor를 사용하도록 한다면, 시스템 내부/외부 모두 아무런 코드 변경없이, 활용할 수 있을 것입니다.

일반적인 Factory Pattern보다 IoC가 더 발전적인 방식인 이유는 Factory Pattern은 내부 시스템에서 약속된 제한된 형식의 Compressor를 제공하는 반면, IoC를 이용하면, 다른 시스템에서 제공하는 ICompressor의 Concrete Class를 구조적으로 활용할 수 있도록 많은 기능을 제공한다는 것입니다. 예를 들어, IoC가 Instancing 뿐 아니라 Instance의 Lifecycle, LifeStyle도 정의할 수 있고, Instance의 속성값도 제공해 줄 수 있습니다.

그러므로 간단한 시스템에서는 Factory Pattern만으로도 충분히 유연한 시스템을 구성할 수 있고, 개방성과 확장성을 고려한다면, IoC 를 통한 구현을 고려하는 것이 좋습니다.

마지막으로 Decorator Pattern을 이용하여 몇가지 Class의 상속과 대비되는 점을 설명하겠습니다.

Decorator pattern에 대해 간략히 설명하자면, 특정 클래스가 하는 일은 방해하지 않고, 그 Class를 Wrapping해서, 부가적으로 다른 일도 수행해 주는 방법을 말합니다. (비슷하지만 Adapter 패턴은 Wrapping한 class를 사용자에게 숨기고, 변형해서 제공하는 패턴입니다.)

가장 많은 예로 드는 것이 FileStream과 BufferedStream 입니다.

만약 파일에 어떤 값을 저장하고자 할 때 다음과 같이 수행합니다.

Using(var fs = new FileStream(filename, accessMode))

{

fs.Write(data);

}

그러면 이 경우는 어떨까요?

Using(var bs = new BufferedStream(new FileStream(filename, accessMode))

{

bs.Write(data);

}

둘 다 파일에 정보를 쓰는 작업을 수행합니다. BufferedStream은 자체적으로 파일(여기서는)에 정보를 쓰는 작업을 수행하는 것이 아닙니다. 내부 코드는 다음과 비슷할 것입니다.

Public class BufferedStream

{

private readonly Stream \_inner;

private byte[] \_buffer;

Public BufferedStream(Stream inner)

{

\_inner = inner;

\_buffer = new byte[default buffer size];

}

Public void Write(byte[] data)

{

Add data to \_buffer.

if \_buffer is full?

{

\_inner.Write(\_buffer);

Clear \_buffer;

}

}

}

예상 했듯이 BufferedStream은 wrapping한 class를 호출합니다. – 물론 사용자로부터 전달된 정보를 변형할 수 도 있지만, 이 예에서는 그렇게 하면 안되겠죠. 어쨌든 BufferedStream은 사용자가 원하는 동작을 수행합니다. 다만 수행 방식에서, 약간의 변화를 준 것이죠. 이름 그래도 버퍼링을 수행합니다.

이런 Decorator pattern에서 일관된 Interface (사실은 Signature가 더 적당한 용어일 것이다) 를 유지하면서, 사용자가 원하는 기본 기능외에 부가 기능을 덪씌워서 작업을 수행할 수 있습니다.

RCL.Data.NH.INHRepository를 이용하여 Decorator pattern을 설명한 자료도 있으니 참고하시면 되구요.

이 경우에도 Signature가 같다는 전제조건이 있는데, BufferedStream의 경우는 Stream이라는 Abstract Class를 기준으로 하였고, NHRepositoryDecorator는 INHRepository라는 Interface를 기준으로 사용하였다는 차이점이 있습니다.

기본적으로는 Interface를 이용하는 것이 더 확장성이 있지만, .NET Framework는 그 자체가 기반이되는 경우라 interface를 정의하지 않고, Base class를 이용하는 것 같습니다.