Magpie
Assembly analysis using Mono.Cecil

Magnus Holmström
Abstract

Magpie

Magnus Holmström

This thesis is about the possibility for program developers to write programs that are able to interact with already managed files. That means that they are compiled into intermediate language files, which is impossible to read for a human.

The objective of this project was to develop a program that would use this possibility to interact with compiled files. This in order to extract information from them about how and if they reference to other chosen compiled files.

This thesis also explains the different tools, such as programs, used in this project to accomplish the task at hand.
1. Introduction

This report describes the path taken when developing an analyze program called Magpie for a company named EPiServer located in Stockholm, Sweden. It reflects the thoughts and choices of the developer as well as it describes the theory and implementation needed to complete the project.

1.1 Background

EPiServer is a Swedish software company that was founded in 1994 and was at that time named ElektroPost Stockholm AB. At that time the company aimed at providing an e-mail system for VIP-clients in Scandinavia, this lead to an intense partnership with Microsoft, a company they are still tightly tied to since EPiServer uses Microsoft’s Visual Studio and C# to develop their platform.

2003 was an important year, since this is when the board decided to streamline EPiServers focus and dismantle their consulting and to convert into a pure software company. The three main areas were now software development, training and hosting, and have stayed the same to this day. [13]

Today EPiServer is one of the leading suppliers of solutions that enable an engaging online presence to increase their customer's business performance. They have more than 4,000 customers all over the world and their platform that combines, online communities, e-commerce and communication, with a great dashboard is the foundation for over 20,000 websites.[1]

1.2 Problem Description

As all other platforms, EPISevers platform needs to evolve in order to fulfill the ever changing requirements from both developers and customers. The problem that complicates it is the fact that at the moment more than 20,000 websites uses EPISevers platform. So even if the developers at EPIServer find parts of code that seams useless or redundant in some way, it's hard for them to know if any of their customers are using that part of the code or not in any of their websites.

If anyone is using that part of the code, it would obviously cause problems if that code would be changed or removed from one version of the platform to the next, unless the developers could in some quick way find out if anyone is using the code and in that case who. If this would be possible the developers at EPIServer would no longer have to guess if it’s safe to
change their code without having to tell all of their customers about the specific changes or not. They could simply change the code in the way they want and contact the concerned customers and tell them about the changes and what they need to do in order to avoid their websites to crash. All the information would be stored in something called an "assembly", an assembly is a file that (in the case of C#) contains the IL (Intermediate Language is text that humans can't read and understand but computers can. If several different programming languages first compile their code to IL code, they can use the same platform and compiler from there on.) representation of the C# code that was compiled.

1.3 Approach

The approach for this project was to start by learning C#. This was done by writing test programs, somewhat like laboratory assignments. The next step was to learn how Mono.Cecil works and how to use it in C#.

When at least a basic understanding about C# and Mono.Cecil had been reached the next was to start working with the structure of the actual Magpie project.

The next step was simply to start programming the different classes of the program and testing them both as a whole with the other classes but also as isolated parts to increase the chance of finding faults or bugs.

1.4 Goals

The goals of this project are quite simple, a program that can find and analyze several already managed assembly files. To find if and how the code within them reference in some way to a chosen signed assembly or assemblies. The program should then store the important information about each reference hit in a way that it’s easy to retrieve, read and analyze in order to solve EPIServers problem about not knowing what parts of their system their customers use.
2. Implementation using C#

This section describes what C# is and how I used it in this project.

2.1 Background

This section contains some amount of background description for C#, the .NET Framework as well as Visual Studio.

2.1.1 C#

C# is a multi-paradigm programming language, this means that it was built to support multiply programming paradigms. In other words, the compiler transforms the written code into IL code (Intermediate Language code) that looks the same for all programming languages. The effect of this is that you may write code in different languages that need to work together and they may do so due to the fact that they are all compiled into the same language before the program is executed.

C# is also an object-oriented programming language, this means that when you write code in C# you always think of each "thing" or "item" as an object with attributes and methods. For example an object may be a car, then you may write a new class Car that has some attributes, color, price, and so on. And also some methods that you may call with a car object to get or change information about the object or something that it is associated with.[2] [14]

2.1.2 .NET Framework

The .NET Framework is an application development platform that is used widely in Visual Studio. It provides services for deploying, running, and building desktop, web and mobile phone applications and services. The first of the two major components in the .NET Framework is a huge class library that provides a lot of reusable code for practically all major areas of application development. The second major component is the common language runtime (CLR), the CLR provides memory management and a bunch of other system services. [15]
The .NET Framework provides eight key services for developers:

- **Memory management** - Usually when developing in most programming languages the programmer is responsible for allocating and releasing memory and thereby the lifetime of different objects and values. In the .NET Framework however, the CLR provides these services to the programmer.

- **Common type system** - In traditional programming languages the compiler defines the basic types. The problem with this approach is that it severely complicates cross-language interoperability. Instead in the .NET Framework the type system defines the basic types which make cross-language interoperability easier since the basic types will be the same for all languages that target the .NET Framework.

- **Vast class library** - By providing a vast class library containing huge amount of code the programmer won't need to write code for common low-level operations.

- **Development frameworks** - The .NET Framework contains additional libraries for specific areas of development such as ASP.NET for web applications development, ADO.NET for specific data access, and more.

- **Language Interoperability** - To make cross-language interoperability the compilers that target the .NET Framework produces a intermediate code called Common Intermediate Language (CIL) that is compiled at run time by the CLR. The result of this is that methods and classes written in one language will be available to other languages at run time, which in turn let programmers develop programs in their preferred language.

- **Version Compatibility** - Close to all applications developed with the .NET Framework can without any modification run on different versions of the .NET Framework. This help the programmer since he won't need to modify his applications each time a new .NET version is available.

- **Side-by-Side Execution** - The .NET Framework prevents version conflicts by allowing multiply versions of the same CLR to run on the same computer. Because of this the same application may also exist on the same computer and execute within the CLR version which it was created.

- **Multitargeting** - It the programmer targets the .NET Framework Portable Class Library it’s possible to create assemblies that work on different .NET Framework platforms, such as the .NET Framework, Silverlight, Windows Phone 7, and even Xbox 360.[3]

All these features of the .NET Framework allows the programmer to choose any programming language that supports the .NET Framework, to develop applications.
Below there is a picture that in a graphical way shows how the CLI, CLR, and the compilers work together within the .NET Framework to enable language interoperability.

![Graphical representation of how the CLI, CLR and the compilers are connected](image)

**Picture 1 - Graphical representation of how the CLI, CLR and the compilers are connected.**[4]
2.1.3 Visual Studio

Visual Studio (VS) is the integrated development environment (IDE) that C# programs are developed in. It has a framework for hosting VSPackages to make it easier to use shared services. VSPackages are software modules that extend the Visual Studio IDE by providing user interface elements, services, editors, and more for the users.[5] An example of this is the user interface that VS provides, it's easy for the user to add buttons and labels using the toolbar and container window. It also provides a good infrastructure that lets the user write code to easily code the user interface.[6]

2.2 Learning C#

Since C# provides a lot of helpful tools the learning curve is very high, in other words it's possible to learn a lot in a short amount of time, and it's quite easy to build big and complicated programs almost instantly. Mainly because of this, the problems that occurred during this project didn't include C# at all. They were more directed towards Mono.Cecil and how and what to search for in the chosen assemblies to find points of reference.

3. Mono.Cecil

This section contains background and usage description about the Mono.Cecil library.

3.1 Background

Probably the most important pieces in this project are the library Mono.Cecil and its sub-libraries, for example Mono.Cecil.Cil. Mono.Cecil and its sub-libraries are a collection of classes and methods that a programmer may use to retrieve information from assembly files. The reason why Mono.Cecil is so important in this project is just its ability to retrieve information from already managed assemblies that contain IL-code (Intermediate Language code). IL-code is otherwise impossible to read with normal tools as the System.IO library that opens streams that one may use to read and write to files. Mono.Cecil also allows a user to modify already managed assemblies on the fly and save the modified assembly back to the disk. There are some classes in the Mono.Cecil library that more or less lay the foundation that all other classes are built upon. The two main classes are the assembly class and the type class.[7]
3.2 The power of Mono.Cecil

The power of Mono.Cecil lies within its ability to retrieve all kinds of information from already managed assemblies, changing the assembly’s data on the fly and store it back to disk. This functionality allows developers to analyze already compiled programs and to do this in a more efficient way. This is also more or less possible to do with another library named Reflection and its sub-library Reflection.Emit. I will, in chapter five, explain what this library is and compare it to Mono.Cecil and explain why Mono.Cecil is more powerful than the built-in Reflection library.

3.3 Learning Mono.Cecil

Mono.Cecil is actually a bit tricky to understand and to figure out how to implement the different functionalities can be time consuming. The reason for this is that there exists no official, or for that matter unofficial, documentation or description of the functionalities of Mono.Cecil or the different classes and their methods. The easiest way to learn Mono.Cecil is actually by searching in forums on the internet where other people write about their problems and how they solved them. Trial and error is also a big part of understanding how Mono.Cecil links different types of objects with each other and how to gain access to one type of object through another type of object.

The assembly class was very important in this project since every time a possible hit was found, that could be a class definition, method parameter of variable type, the program needs to checked if the assembly definition of their declaring type was the correct assembly. The type class was also very important since practically everything analyzed was in some way located within a type definition object, this includes things like; methods, enumerators, attributes, objects, variables and so on.
3.4 Problems with Mono.Cecil

The main problem with Mono.Cecil is the lack of documentation of the different methods and objects that are encapsulated in Mono.Cecil. This is mainly because Mono.Cecil is still in development and its structure isn’t fully locked-in yet. The project team is still adding new functionalities and changing the different API’s (Application Programming Interface – Specification of the interface that different software components uses to communicate) of Mono.Cecil between different versions and updates.

4. XML

This section contains some background and usage description about XML.

4.1 Background

Extensible Markup Language (XML) is a markup language that by defining a set of rules for document encoding, creates a format that can be read by humans as well as computers. The main design goal of XML is usability, generality and simplicity, especially over the Internet. XML is a textual data format and due to the fact that it is used all over the world it has a strong support via Unicode to be able to represent all needed languages. The main usage of XML is representation of data structures, for example in web services.[8]

By today's date, many hundreds of XML-based languages have been developed, some popular examples are RSS, SOAP, and XHTML. XML-based formats have become the default format base for many of the top office tools as LibreOffice, OpenOffice, Apple's iWork, and Microsoft Office.[9]

The definition of an XML document excludes texts that don’t satisfy a special list of syntax rules, some of the key rules are:

- The document may only contain properly encoded Unicode characters.
- The element tags are case-sensitive and may not contain any of the following characters: !, "', $, %, &, (, ), *, +, , , /, ; , <, =, >, ?, @, [, \, ], ^, `, {, |, }, ~.
- There is always a single root element that contains all other elements.

If the XML document satisfies all syntax rules the document is "Well-formed", if it doesn’t, the XML processor is obligated to cease normal processing and report the violation as an error. This method where the execution is terminated when a fault is found it is referred to as "draconian", which means that it may be a bit harsh in some cases. And XML’s policy in
this area has been criticized since it violates Postel's law - "Be conservative in what you send, be liberal in what you accept".[8]

The XML design goals include the fact that it should be easy to write programs that process XML documents as both input and output. Despite this fact, the XML specification contains almost no information about how programmers are supposed to go about when using XML documents as recourses for their programs. A number of APIs (Application Programming Interface) has been developed and some even standardized. Each of these APIs for XML processing falls into one of the following categories:

- **Stream-oriented APIs** accessible from a programming language when using streams to handle data, example: StAX.
- **Tree-traversal APIs** accessible from a programming language when using some type or tree to store and use date, example: DOM.
- **XML data binding**, which provides automated translations to and from an XML document and programming-language objects.
- **Declarative transformation languages**, example: XQuery. [9]

Even if Tree-traversal APIs tend to use more memory than APIs from other categories, it's easier and more intuitive and therefore more convenient for programmers to use. This is also the type of approach I choose as I constructed my XML document as an XML-tree. [9]

Despite the functionality that XML adds to programming languages it's been widely criticized for its complexity. The main reason for its complexity is the fact that it's used for exchanging highly structured data between different applications even though this was not its primary design goal. The problem here is that the mapping from a basic tree model of XML to the type systems of programming languages is really difficult to accomplish. [10]
4.2 Using XML

Using XML was quite straightforward actually, to create an XML tree and fill it with the desired info about all references that the program found only a few commands was needed. The first command used was to create a new element (XElement in XML) that was named "Hits", then for each hit. The same command was used to create a new element but this time with a couple of attributes (XAttribute in XML), these attributes is where the actual information is stored and represented in the XML tree. Then the new attribute was added as a child to the first element "Hits", this was the repeated for each reference hit that was found during the analysis faze. When all reference hits have been added to the tree as child nodes to the first element "Hits" the "Save" command was used to store the tree in a file at the location specified by the user.

This is the actual C# code used for the different commands:

- Create the first element "Hits": XElement Hits = new XElement("Hits");
- Create a new element with attributes and add it to the XML tree:
  Hits.Add(new XElement("Hit", new XAttribute("HitType", hitType), ..(some additional attributes).., new XAttribute("ReferencedMember", referenceMem)));
- Save the XML tree to a file: Hits.Save(xmlFile); (Where xmlFile is the filepath chosen by the user).

5. The Reflection library

This section contains some background description about the Reflection library in the .NET Framework as well as a comparison between the Reflection library and Mono.Cecil.

5.1 Background

Reflection is a already existing library located in the .NET Framework. The goal of the Reflection library is the same as that of Mono.Cecil, to observe and modify the structure and behavior of a certain program. Or in simpler words, to allow the user or a program to retrieve and store information in already managed assemblies.[11]

As with Mono.Cecil there are some classes that lay the foundation of the library, in Reflection this is mainly done by the Type class. Since all classes inherits from the Object class, the Type class serves as runtime information about the assembly, the module or even the type.[12]
5.2 Reflection vs. Mono.Cecil

Most of the functionality found in the Mono.Cecil library also exists in the Reflection library, although the main difference between the two libraries and a crucial part for me in this project is Mono.Cecil's ability to retrieve information about single instruction in methods. It is in other words possible with Mono.Cecil to retrieve information about local variable definitions and usage among many things. This was of course a very important part in this project since instructions also contains method calls which is the single most important part to analyze.

6. The Execution

In this section I present how I went about planning and executing this project.

6.1 Planning

The first step, planning, was without a doubt very important since it was during this step the structure was formed. After a couple of tries, the structure below is the chosen one. It is easy to add new classes without having to change the already existing code.

The main classes were:

- **GUI (Graphical User Interface)** - This class handles all communication with the user via the GUI that was created using Visual Studios built-in functions.
- **FileFinder** - This class is responsible for locating and storing the paths to all relevant files (in this project, all .dll files) in a folder given by the user and all its sub-folders.
- **Analyzer** - This is the most complex class that is responsible for the complete analysis of all files found by the FileFinder class. The analysis part includes both analysis of each file to find all point of reference from that file to a file with the right signature, that is also given by the user, and storing the found results.
- **Signature** - This class is responsible for all things connected to the signature of assemblies. The signature of an assembly is something that programmers can use as a way to show what company, person or project a certain assembly belongs to. In this project only two main methods was needed for signature handling:
  - **Signature Converter** - This method converts a string of length 16 into a byte array of length eight. This to be able to easier compare the signature given by the user with the signatures that is extracted with Mono.Cecil.
- **Signature Comparer** - This method simply compares the two byte array signatures to check if a point of reference was found or not.

### 6.2 GUI

Before any piece of code was written, the graphical user interface (GUI) was created. This to make it easier to test the code and make it easier to see what methods to start with to be able to use the GUI to test more complex methods later on. Below there is a snapshot of the simple GUI, since the information found was to be stored in an XML file there was no need for a more complex GUI that would include a way to display the found reference hits. The different parts of the GUI are explained below and shown in picture two below the explanations:

- **Folderpath** - In this textbox the user types the path to the folder he or she wants the program to search in and analyze.
- **Reference Signature** - In this textbox the user types the 16 character assembly signature that he or she wants the program to match references to.
- **Storage File** - In this textbox the user types the path to the file he or she wants the program to store all found references hits in XML form.
- **Information** - In this textbox information is printed during the execution of the program. For example the first thing that is printed when the analyze button is pressed is "Searching for assemblies...".
- **Analyze** - This button initiates the analysis phase of the program.
- **Clear** - This button clears the text from all textboxes.
- **Quit** - This button simply shuts down the application.
6.3 Analyzer

In this section I present how I created the file finder and the analyzer class, which are both part of the analyzer section of the program.

6.3.1 File Finder

The file finder class is a quite small class that only contains a few methods, still it handles both file finding and storing as well as checking if paths given by the user are correct. The first thing I did was the method that checks if the folder the user want the program to search for files to analyze exists or not. Thanks to the Directory library in the .NET Framework, this was easily done with just one command. The next step was to write the code that would search the given folder path and all sub folders in that folder and so on for .dll files (assembly files) that's not signed with the given signature. This is to avoid self references.

To do this three methods was required, the first method "findFiles" is responsible for creating the needed list to store the file paths of found .dll files and call the other methods to get things started. The second method "browseFolders" responsibility is for each found folder in the given folder call itself in order to browse all sub-folders, but also call the
The "getFiles" method checks all the files in the given folder for .dll files with a different signature than the reference signature given by the user and store the path to those files in the list created by the first method.

When all folders have been scanned and all matching files found, the file list is returned to the method that first called the "findFiles" method.

Below is the actual code for both the "browseFolders" and the "getFiles" methods. The exceptions cast in the "getFiles" method is to prevent the program from crashing when a file can't be opened because it's protected or that it's of the wrong file type.

```csharp
private void browseFolders(string folderPath, ref List<string> fileList, byte[] byteSigRef)
{
    string[] foldersTemp = Directory.GetDirectories(folderPath);
    foreach (string folder in foldersTemp)
    {
        getFiles(folder, ref fileList, byteSigRef);
        browseFolders(folder, ref fileList, byteSigRef);
    }
}
```
private void getFiles(string folder, ref List<string> fileList, byte[] byteSigRef)
{
    string[] filesTemp = Directory.GetFiles(folder);
    foreach (string file in filesTemp)
    {
        if (file.EndsWith(".dll"))
        {
            try
            {
                AssemblyDefinition assemblyDefinition = AssemblyFactory.GetAssembly(file);
                if (!signature isEqual(byteSigRef, assemblyDefinition.Name.PublicKeyToken))
                {
                    fileList.Add(file);
                }
            }
            catch (UnauthorizedAccessException)
            {
                System.Windows.Forms.MessageBox.Show("Attempt to access " + file + "
                denied!");
            }
            catch
            {
                System.Windows.Forms.MessageBox.Show("Attempt to access " + file + "
                failed!");
            }
        }
    }
}

6.3.2 Analyzer

The analyzer is the single most complex and important part of the entire project. This is where each and every found file gets analyzed from the outside and in. In more technical terms, the analysis starts at the assemblies’ definition and works its way down through all the layers of object definition and their contained objects.

The picture below shows in a very simplified way how the structure looks that Mono.Cecil implements. The light blue boxes represent object definitions and the red boxes represent attributes that are important to check since they may reference correctly signed assemblies.
Once again this is a simplified version of the real structure, but that structure would never fit a normal sized paper, and it wouldn't be easy to understand either.

![Diagram of assembly definition structure](image)

*Picture 3 - Shows a simplified version of the structure that Mono.Cecil implements.*

As you can see in picture three, to be able to access all variable definitions you first need to access all method definitions and so on up through the tree like structure in the picture above. The problem with this is that there is no efficient way to analyze all variable definitions or similar objects without looping through all assembly-/class- and method definitions. This of course increases the total time complexity of the program, but there is really no way to get around this problem, the only solution is the “brute force” implementation that I chose.

### 6.3 XML

Since the access to EPiServers database was closed another way was needed to store all the information about all the reference hits found by the program. The easiest and yet very efficient way was to store all the information in an XML document. This allows the programmers on EPiServer to retrieve the information easy and store it in their database for further usage. The structure of the XML tree was very simple, one root node named "Hits" with children nodes that each represent one reference hit. The important information about each hit is stored in attributes connected to the children nodes.
The attributes of any given child node are the following:

- **HitType** - What kind of type the hit is, for example: "BaseClass", "Variable", and "MethodParameter".
- **ReferringApplication** - The path to the application that the assembly belongs to that the hit was found in.
- **ReferringAssembly** - The assembly the hit was found in.
- **ReferringType** - If the hit was found within a class, this attribute is that classes name, otherwise it's null.
- **ReferringMember** - If the hit was found within a method, this attribute is that methods name, otherwise it's null.
- **ReferencedAssembly** - The assembly the hit reference to.
- **ReferencedType** - If the hit referenced to a class or something within a class, this attribute is that classes name, otherwise null.
- **ReferencedMember** - If the hit referenced to a method or something within a method, this attribute is that methods name, otherwise null.

This part of the code was actually pretty easy to write, but time consuming since each type of hit needed its own method that added that hit to the tree. This is because the information can’t be retrieved in the same way for each type of hit, rather the opposite. The text below shows a typical hit that is stored in the XML document. This specific hit is a "BaseClass" hit, that means that the class "EPiServerWorld.Global" inherits from the class "EPiServer.Global" that is defined in the assembly "EPiServer". The "EPiServer" assembly has the PublicKeyToken (signature) that the program was looking for when this hit was found.

```
<Hit HitType="BaseClass"
     ReferringApplication="S:\Users\Magnus\Documents\Visual Studio 2010\Projects\EpiServer"
     ReferringAssembly="EPiServerWorld, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
     ReferringType="EPiServerWorld.Global"
     ReferringMember="null"
     ReferencedAssembly="EPiServer, Version=6.1.379.0, Culture=neutral, PublicKeyToken=8fe83dea738b45b7"
     ReferencedType="EPiServer.Global"
     ReferencedMember="null" />
```
6.4 Testing

This part was kind of tricky but really important and a great opportunity to learn the more tricky and advanced ways of how to write code in C#. In order to be able to fully test the program the test code needed to be written in two separate projects in order for the code to be compiled into two different .dll files.

The code in the first project (project A) was to be compiled into the assembly that the program would search for references to. Because of this, all the code in project A is about defining types, attributes, enumerators and classes that could be referenced to from another class or project, in other words, the code in project A doesn't really do anything. In the other project (project B), a couple of classes was written with code that would in every way possible, in C#, reference to a object that is defined in project A. Some types of this code could look like this:

- **ClassA aClassAObject = new ClassA();** - This created a new object of type ClassA that is located in project A. The "aClassAObject" will therefore be found by the program as a reference point.

- **public class ClassC : ClassA** - This is the first line when creating a new class "ClassC", the ": ClassA" part means that "ClassC" inherits from "ClassA". This will also be found by the program as a reference point.

The whole testing part worked great and it showed that the program found everything it was suppose to find, of course this doesn't mean there's not any bugs, but at least it works according to the goals that was set up at the beginning of the project.

7. Concluding Remarks

This section is about the positive and negative aspects of this project and what could have been done better. The last part is some final thoughts about how well the project went and what was the most interesting parts.

7.1 The Positive

The fact that the project was finish and could be delivered as a working program to EPiServer that they will be able to use when developing new versions of their platform is without a doubt the most positive aspect of the entire project.
A positive thing about the execution of the program is that even when analyzing larger pieces of code for entire websites that produces thousands of reference hits and considering that the analyze part of the code is “brute force”, the program only runs for a few seconds before the entire code is analyzed and the result is stored in the desired xml document.

7.3 Improvements

One improvement that’s less important for the efficiency of the program but very important for code maintainability is to structure the code even more. Even if the code is already structured in such a way that each part of the program that handles one specific thing, for example analyze of variable types, has its own method. In other words, every method handles only one part of the program. An improvement to this would be to gather the methods that handle similar parts of the program and group them together in some way, either just put them all close together or even put them in an own class or sub-class.

Another improvement that would enhance the scalability of the code, and therefore make it easier to add new items to analyze, would be to implement an interface. This interface would handle polymorphic lists that all reference hits would be stored in. This would greatly reduce the size of the class that handles the storing of information to the xml document, since each type of hit needs an own method due to the way the storing mechanism was implemented. If the interface would have been implemented, one method could handle all kinds of objects and therefore improve the scalability of the code.

7.4 Last Thoughts

This project has without a doubt been a great learning experience, both from a programmers and a developer’s point of view. I’ve learned the importance of well executed planning and how it improves the quality of the code and also the program itself in the aspects of maintainability and efficiency during runtime. I also believe that I’ve improved as a programmer, mainly because this is the first project that I've done on my own and this has forced me to learn everything that was needed to be able to complete this project.
8. References

8.1 Electronic Sources

1. EPiServer, *Om oss*, episerver.se  
Availability: [http://www.episerver.se/Om-oss/](http://www.episerver.se/Om-oss/)  
Retrieved: 2012-05-10

2. C# Language Specification  
Availability: [C# Language Specification](#)  
Retrieved: 2012-04-25

3. MSDN  
Retrieved: 2012-04-26

4. CLI Structure  
Retrieved: 2012-04-26 11.39

5. VSPackages  
Retrieved: 2012-04-25

6. Visual Studio Development Model  
Retrieved: 2012-04-25

7. Mono, *Cecil*, mono-project.com  
Availability: [http://www.mono-project.com/Cecil](http://www.mono-project.com/Cecil)  
Retrieved: 2012-04-25

8. W3  
Availability: [http://www.w3.org/TR/REC-xml/](http://www.w3.org/TR/REC-xml/)  
Retrieved: 2012-05-18

9. XML  
Availability: [http://xml.coverpages.org/xmlApplications.html](http://xml.coverpages.org/xmlApplications.html)  
Retrieved: 2012-05-18
10. XML Another Angle
Retrieved: 2012-05-18

11. C# Tutorial
Retrieved: 2012-06-04

12. C# Tutorial Types
Availability: [http://csharp.net-tutorials.com/reflection/the-right-type/](http://csharp.net-tutorials.com/reflection/the-right-type/)
Retrieved: 2012-06-04

13. EPiServer, Historik, episerver.se
Availability: [http://www.episerver.se/Om-oss/Historik/](http://www.episerver.se/Om-oss/Historik/)
Retrieved: 2012-05-10

14. C# for Students - Douglas Bell, Mike Parr
ISBN: 978-0321-176653

15. .NET Compact Framework Programming with C# - Paul Yao, David Durant
ISBN: 0321174038