CrypTool 2.0

Plugin Developer Manual

– How to build your own plugins for CrypTool 2.0 –

S. Przybylski, A. Wacker, M. Wander, F. Enkler and P. Vacek

{przybylski|wacker|wander|enkler|vacek}@cryptool.org

Version: 0.5

February 24, 2010

CrypTool 2 is the modern successor of the well-known e-learning platform for cryptography and cryptanalysis CrypTool 1, which is used world-wide for educational purposes at schools and universities and in companies and agencies.

Since the first launch of CrypTool 1 in 1999 the art of software development has changed dramatically. The CrypTool 2 team began working in 2008 to develop a completely new e-learning application, embracing the newest trends in both didactics and software architecture to delight the end-user with an entirely new experience.

CrypTool 2 is built using

- .NET (a modern software framework with solutions to common programming problems from Microsoft),

- C# (a modern object-oriented programming language, comparable to Java), and

- WPF (a modern purely vector-based graphical subsystem for rendering user interfaces in Windows-based applications), plus

- Visual Studio 2008 (a development environment) and

- Subversion (a source code and documentation version management system).

This document is intended for plugin developers who want to contribute new visual or mathematical functionality to CT2. As of January 2010, the code consists of about 7000 lines of C# code in the core system and about 240,641 lines of C# code in 115 plugins.

For further news and more screenshots please see the developer page http://www.cryptool2.vs.uni-due.de.

Contents

List of Figures

1 Developer Guidelines

CrypTool 2.0 uses state-of-the-art technologies like .NET 3.5 and WPF. In order to make your first steps towards developing something in the context of this project, a few things need to be considered. First of all, please follow the instructions in this document so that you do not get stuck. If you encouter a problem or error that is not described here, please let us know so we can add the appropriate information to this guide.

In the following sections we will describe all steps necessary in order to compile CrypTool 2.0 on your own. This is always the first thing you need to do before you can begin developing your own plugins and extensions. The basic steps are:

- Getting all prerequisites and installing them

- Accessing and downloading the source code with SVN

- Compiling the current source code for the first time

1.1 Prerequisites

Since CrypTool 2.0 is based on Microsoft .NET 3.5, you will need a Microsoft Windows environment. (Currently no plans exist for porting this project to mono or to other platforms.) We have successfully tested with **Windows XP**, **Windows Vista** and **Windows 7**.

Since you are reading the developer guidelines, you probably want to develop something. Hence, you will need a development environment. In order to compile our sources you need **Microsoft Visual Studio 2008 Professional**. Please always install the latest service packs for Visual Studio. Unfortunately, our sources do not work (smoothly) with the freely available Visual Studio Express (C#) versions. This is due to the fact that CrypWin uses a commercial component and is therefore distributed only as binary, and the current version of C# Express cannot handle a binary as a start project, which makes debugging cumbersome. This will be resolved later in 2010 when the project is moved to Visual Studio 2010.

Usually the installation of Visual Studio also installs the .NET framework. In order to run or compile our source code you will need (at the time of writing) at least **Microsoft .NET 3.5 with Service Pack 1 (SP1)**. You can get this for free from Microsoft's webpage. Once that has been installed, your development environment should be ready for our source code.

1.2 Accessing Subversion (SVN)

Now you will need a way of accessing and downloading the source code. In the CrypTool 2.0 project we use Subversion (SVN) for version control, and hence you need an **SVN client**, e.g. **TortoiseSVN** or the **svn commandline from cygwin**. It does not matter which client you use, but if you have never worked with SVN before, we suggest using TortoiseSVN, since it offers a nice Windows Explorer integration of SVN.

The CrypTool2 SVN URL

Our code repository is accessable at the following URL:

https://www.cryptool.org/svn/CrypTool2/

To access the repository, you must provide a username and password. If you are a guest and just want to download our source code, you can use "anonymous" as the username and an empty password. If you are a registered developer, just use your provided username and password (which should be the same as for the wiki).

Accessing the repository with TortoiseSVN

As mentioned above, in order to access the SVN repository one of the best options is TortoiseSVN. We will describe here how to use the basics of the program, although you should be able to use any SVN client in a similar fashion.

First install TortoiseSVN (which unfortunately requires you to reboot your computer) and then create a directory (for instance "CrypTool2") for storing the local working files somewhere on your computer. Right-click on this directory and select "SVN Checkout" from the context menu. A window will appear in which you will be asked for the URL of the repository as given above. The "Checkout directory" should already be filled in correctly with your new folder. Then just hit "OK", accept the certificate (if necessary), and enter your login information as described above. Mark the checkbox for saving your credentials if you don't want to enter them every time you work with the repository. Then hit "OK", and now the whole CrypTool2 repository should be checked out into your chosen directory.

Later on, if changes have been made in the repository and you want to update your working copy, you can do this by right-clicking on any directory within the working files and choosing "SVN Update" from the context menu. You should do this often to maintain a current version of the files.

A TortoiseSVN tutorial can be found here.

Committing your changes

If you are a registered developer, you can commit your file changes to the public CrypTool2 repository. Right-click on the directory within the working files that contains your changes and select "SVN Commit" from the context menu to upload your changes. Please always provide *meaningful descriptions* of your updates. You should commit your sources to our SVN repository as often as you can to ensure your interoperability with the rest of the project, but only commit code that successfully compiles and runs!

You can use command words in the SVN comment to link your changes to a particular ticket. The command syntax is as follows:

```
command #1
command #1, #2
command #1 & #2
command #1 and #2
```

You can have more than one command in a message. The following commands are supported. There is more than one spelling for each command, to make this as user-friendly as possible.

```
closes, fixes:
The specified issue numbers are closed with the contents of this commit
message being added to it.
references, refs, addresses, re:
The specified issue numbers are left in their current status, but the
contents of this commit message are added to their notes.
```

A fairly complicated example of what you can do is with a commit message of:

```
Changed blah and foo to do this or that. Fixes #10 and #12, and refs #12.
```

This will close #10 and #12, and add a note to #12.

Ignore patterns

Please only check in proper source code by using the following **ignore patterns**:

obj bin debug release *.pdb *.suo *.exe *.dll *.aux *.dvi *.log *.bak *.bbl *.blg *.user

This basically means that you should never check in compiled and automatically generated files. For example, please do not check in the entire *bin/* and *obj/* directories that Visual Studio generates. Note that the server will reject your commits if you try to do so. If you want to submit a component (binary file) despite the ignore patterns you can still add **.dll* files by using the context menu and adding the file explicitly - but please be absolutely sure that you know what you are doing. Additionally, you need to provide an explicit list of file and directory names which should override the ignore pattern. For example, if you want to check in a file named someLib.dll, you must write a comment which looks like this:

```
The lib is required by all developers, so I am adding it explicitly to the
repository.
override-bad-extension: someLib.dll
```

Please note that any text after the colon and the whitespace will be treated as the file name. Therefore, do not use quotation marks and do not write any text after the file name.

1.3 Compiling the sources

By this point you should have checked out a copy of the entire CrypTool repository. Compiling is pretty easy; just go to the *trunk/* directory and open the **CrypTool 2.0.sln** Visual Studio solution. The Visual Studio IDE should open with all the working plugins components nicely arranged. In case you are now starting Visual Studio for the first time, you will have to choose your settings. Just select either "most common" or "C#" — you can change this at any time later. On the right side is the project explorer, where you can see all the subprojects included in the solution. Look for the project **CrypWin.exe** there. Once you have found it, right-click on it and select "Set as StartUp-Project" from the context menu. Next, go to the menu bar and select "Build" → "Build Solution".

Then go to "Debug" and select "Start Debugging". CrypTool 2.0 should now start for the first time with your own compiled code. Presumably you have not changed anything yet, but you now have your own build of all the components (with the exception of CrypWin and AnotherEditor, since they are available only as binaries). If the program does not compile or start correctly, please consult our FAQ and let us know if you found a bug.

If you are a **core developer**, hence somebody who can also compile CryWin and AnotherEditor, you should use the **CrypTool 2.0.sln** solution from the *trunk/CoreDeveloper/* directory (which will *not* be visible to you if you are not a core developer). As a core developer, be aware that when you compile, you **change the CryWin.exe** that is visible to everybody else. Thus, when doing a check-in, please make sure you *really* want to check in a new binary. Core developers can also build a new setup and publish it as beta release on the website. This process is explained in the wiki at https://www.cryptool.org/trac/CrypTool2/wiki/BuildSetup.

2 Plugin Implementation

In this chapter we provide step-by-step instructions for implementing your own CrypTool 2.0 plugin. The given instructions refer mostly to the usage of the Visual C# Express and Visual Studio Professional 2008 editions, so before starting you should have a copy of **Microsoft Visual Studio 2008** (or **Microsoft Visual C# 2008 Express Edition**) installed on your computer. We will use the **Caesar cipher** (also known as the **shift cipher**) for our example implemenation.

2.1 Creating a new project

To begin, open Visual Studio, go to the menu bar and select "File" → "New" → "Project...". The following window will appear:
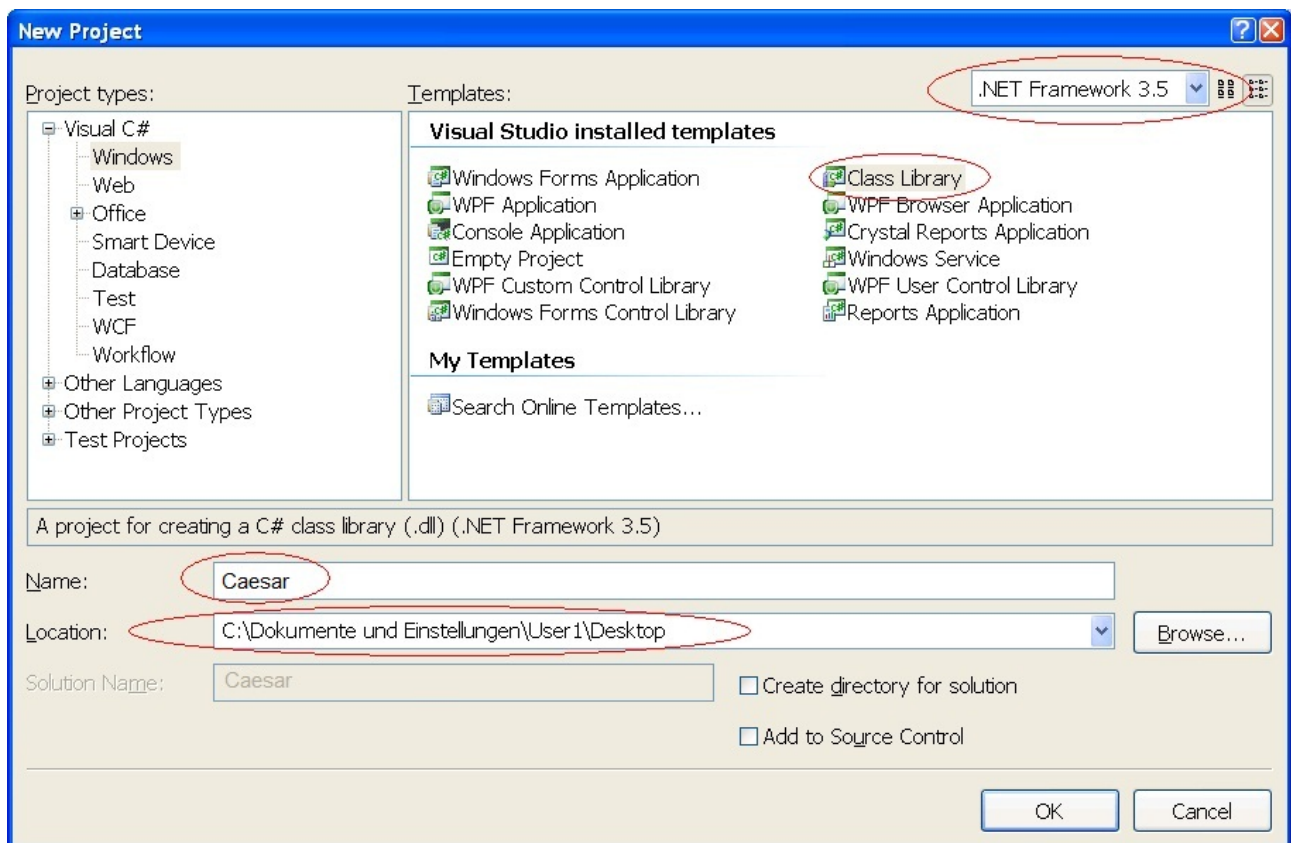
Figure 2.1: Creating a new Visual Studio project.

If you are using Visual Studio 2008, select **".NET-Framework 3.5"** as the target framework; the Express Edition will automatically choose the target framework. Then choose **"Class Library"** as the default template, as this will build the project for your plugin as a DLL file. Give the project a unique and meaningful name (such as "Caesar" in our case), and choose a location to save it to. (The Express Edition will ask for a save location later when you close your project or environment). Select the subdirectory "CrypPlugins" from your SVN trunk as the location. Finally, confirm by pressing

the "OK" button. Note that creating a new project in this manner also creates a new solution into which the project is placed.
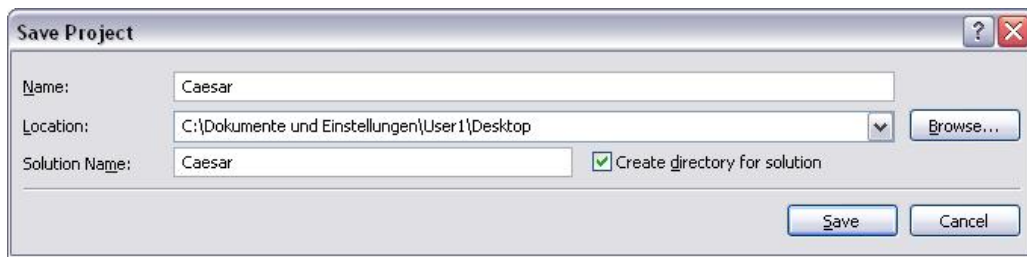
Figure 2.2: The Microsoft C# Express Edition "Save Project" dialog window.

At this point, your Visual Studio/C# Express solution should look like this:
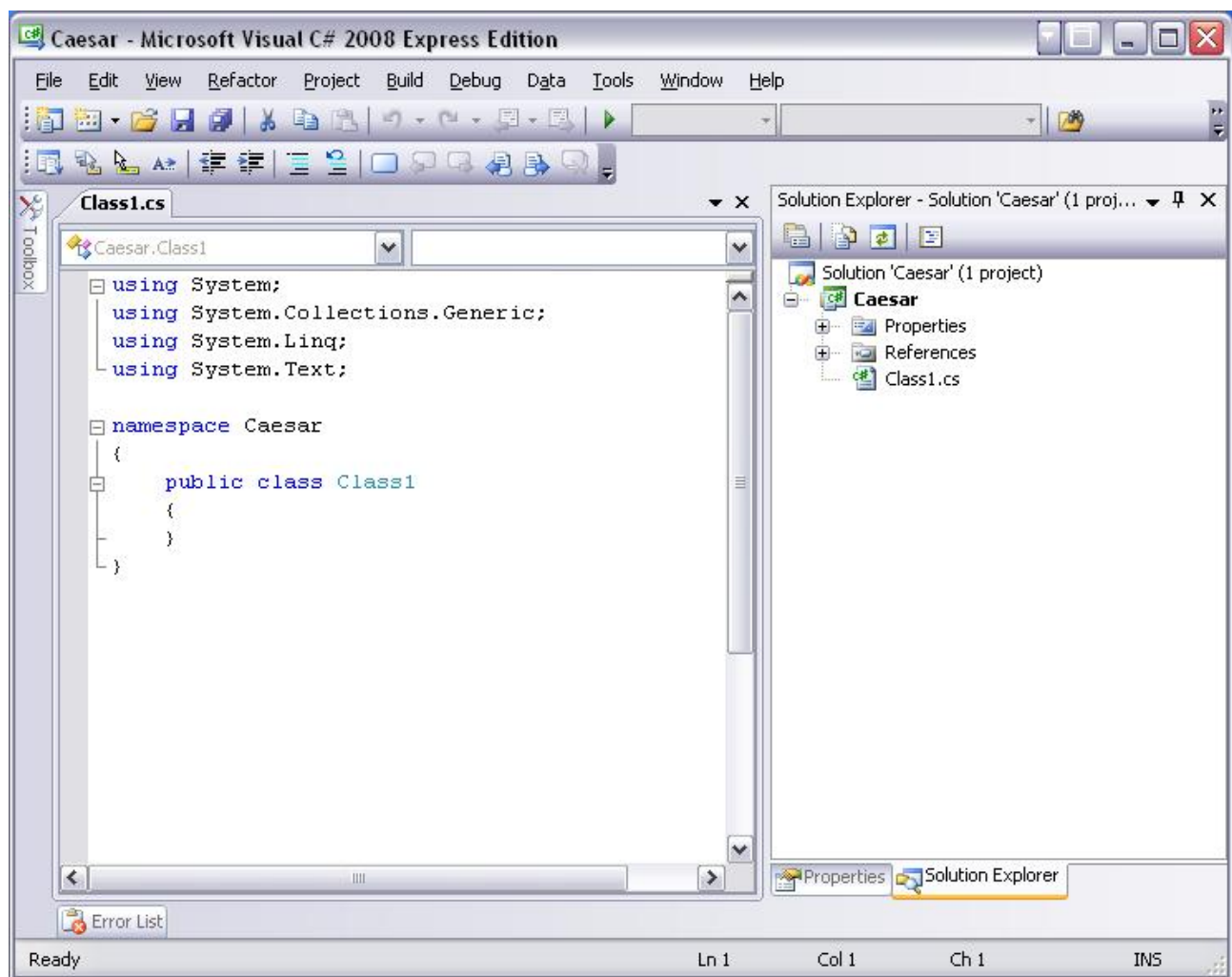
Figure 2.3: A newly created solution and project.

2.2 Interface selection

To include our new plugin in the CrypTool program, we must first add a reference to the CrypTool library, **CrypPluginBase.dll**, where all the necessary CrypTool plugin interfaces are declared.
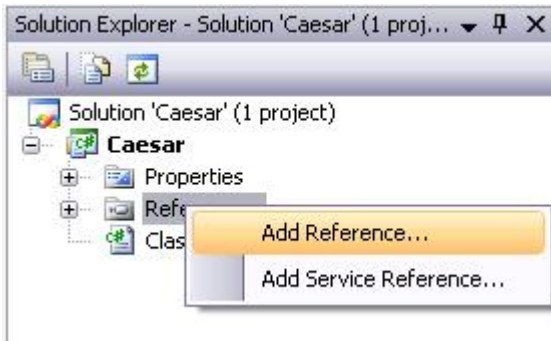
Figure 2.4: Adding a new reference.

Right-click in the Solution Explorer on the "Reference" item and choose "Add Reference". A window like the following should appear:
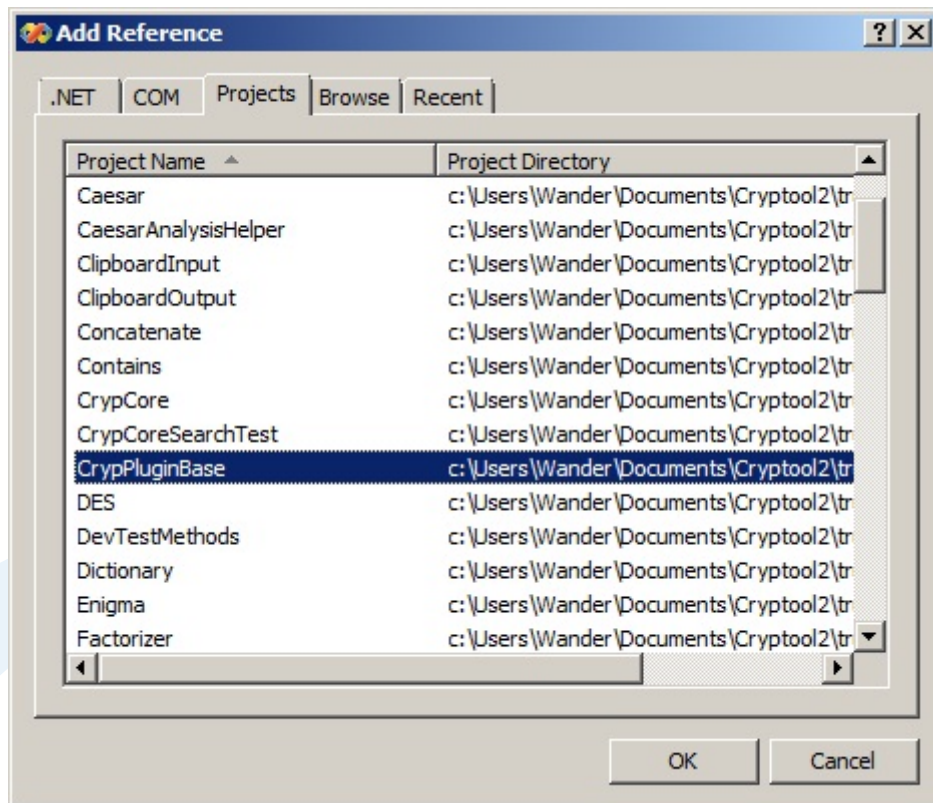
Figure 2.5: Adding a reference to the PluginBase source code.

Select the project "CrypPluginBase". If you do not have the "CrypPluginBase" source code, it is also possible to add a reference the binary DLL. In this case browse to the path where the library file *CrypPluginBase.dll* is located, e.g. *C:\Documents and Settings\<Username>\My Documents\Visual Studio 2008\Projects\CrypPluginBase\bin\Debug* and select the library by double clicking the file or pressing the "OK" button. (You can also select the binary DLL located in the folder where *CrypWin.exe* was placed when you downloaded CrypTool2.)
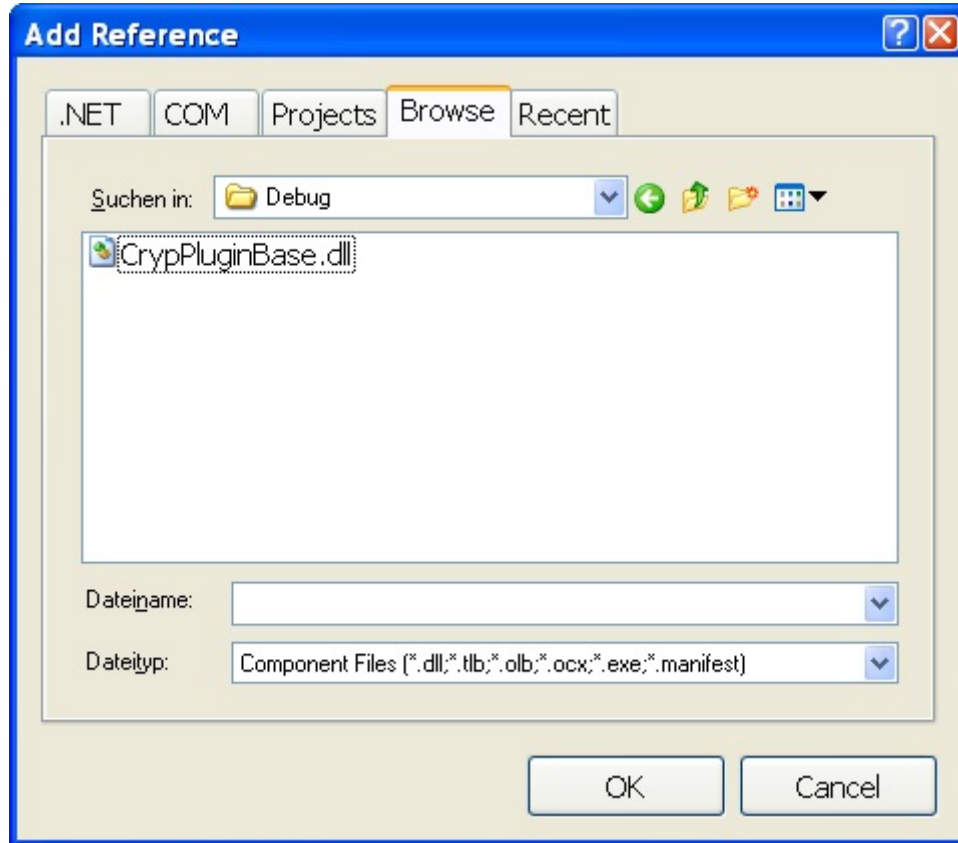
Figure 2.6: Browsing for a reference.

Besides CrypPluginBase you will need to add three assembly references to provide the necessary "Windows" namespaces for the **user control** functions "Presentation" and "QuickWatchPresentation". This can be done in the same manner as before with the "CrypPluginBase" but by selecting the ".NET" tab. Select the following .NET components:

- PresentationCore

- PresentationFramework

- WindowsBase

Afterwards your reference tree view should look like this:
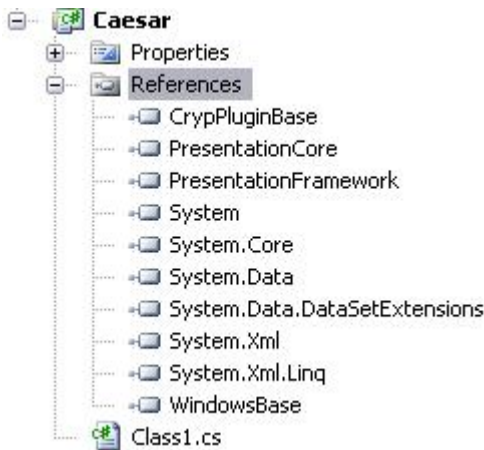
Figure 2.7: A reference tree with the essential components.

If your plugin will be based on other additional libraries, you can add them in the same way.

2.3 Modifing the project properties

It is important to make two small changes to your plugin's assembly data to make sure that it will be imported correctly into CrypTool 2. Go to the Solution Explorer and open "AssemblyInfo.cs", which can be found in the "Properties" folder. Make the following two changes:

- Change the attribute "AssemblyVersion" to have the value "2.0.*", and

- Comment out the attribute "AssemblyFileVersion".

This section of your assembly file should now look something like this:

```
1 [assembly: AssemblyVersion("2.0.*")]
2 //[assembly: AssemblyFileVersion("1.0.0.0")]
```

2.4 Creating classes for the algorithm and its settings

In the next step we will create two classes. The first class will be the main driver; we will call ours "Caesar" since that is the name of the cipher that it will implement. In our case, this class has to inherit from IEncryption because it will be an ecryption plugin. If it was instead a hash plugin, this class should inherit from IHash. The second class will be used to store setting information for the plugin, and thus we will name ours "CaesarSettings". It has to inherit from ISettings.

2.4.1 Creating a class for the algorithm

When starting a new project, Visual Studio automatically creates a class which has the name "Class1.cs". Since this is a rather non-descriptive name, we will change it. In our example, it should be "Caesar.cs". There are two ways to change the name:

- Rename the existing class, or

- Delete the existing class and create a new one.

Both options will achieve the same results. We will guide you through the second method. First, delete "Class1.cs".
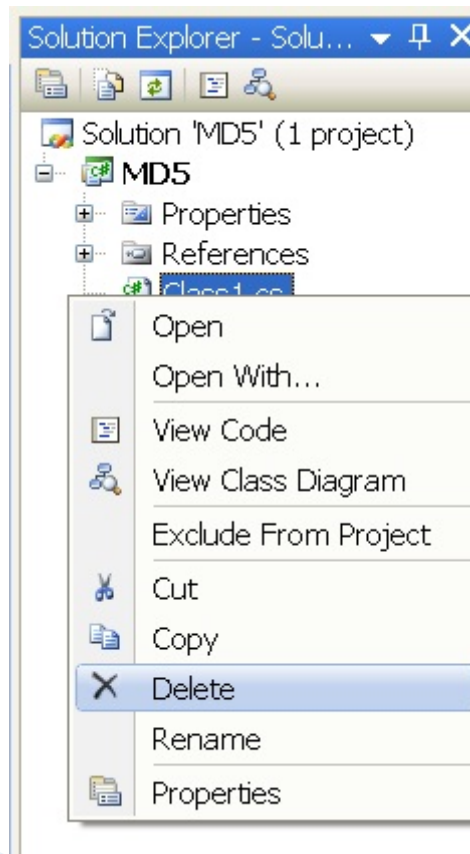
Figure 2.8: Deleting a class.

Then right-click on the project item (in our case, "Caesar") and select "Add → Class. . . ":
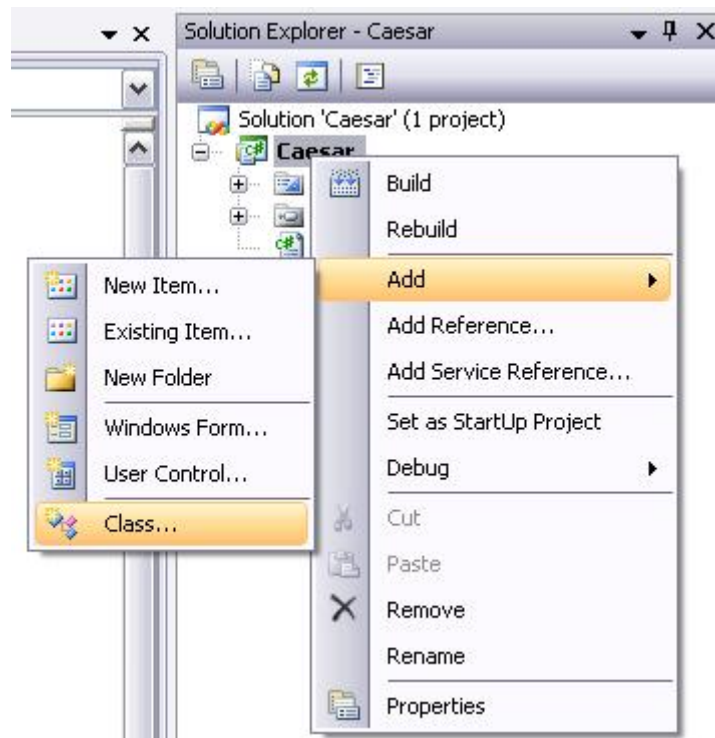
Figure 2.9: Adding a new class.

Finally, give your class a unique name. We will call our class "Caesar.cs" and define it as public so that it will be available to other classes.
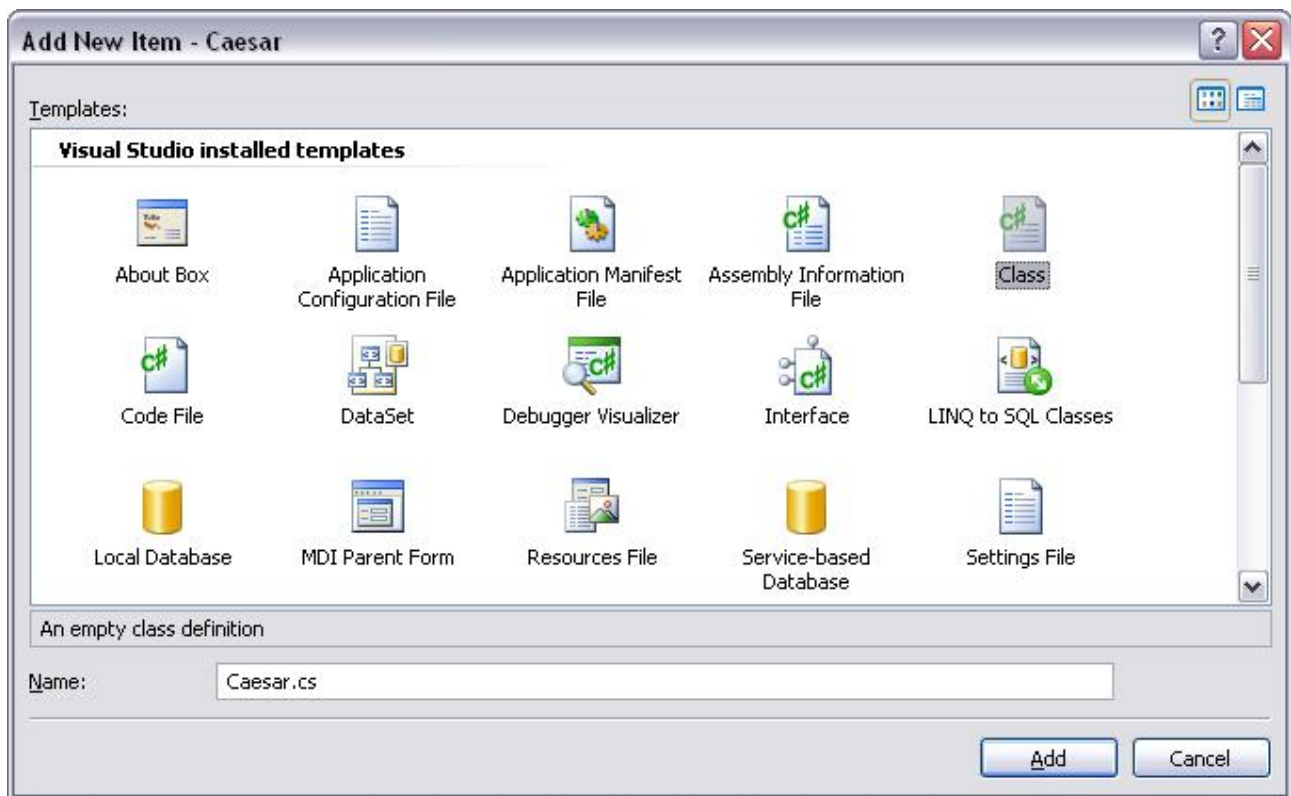
Figure 2.10: Naming the new class.

Visual Studio will automatically generate a basic code outline for the new class. In our example, we will not use the all the namespaces that are automatically imported, so you can delete the lines `using System;` and `using System.Linq;`.

2.4.2 Creating a settings class

Add a second public class in the same way. We will call the class "CaesarSettings". The settings class stores the necessary information about controls, captions, descriptions and default parameters (e.g. for key settings, alphabets, key length and type of action) to build the **TaskPane** in the CrypTool application.

Below is an example of what a completed TaskPane for the existing Caesar plugin in CrypTool 2 looks like:
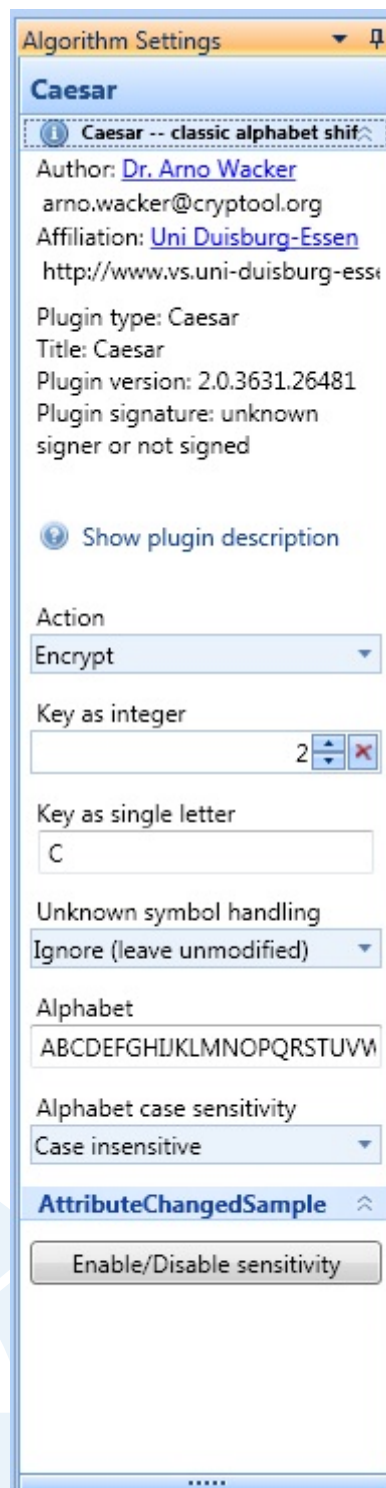
Figure 2.11: The completed TaskPane for the existing Caesar plugin.

2.4.3 Adding the namespaces and inheritance sources for the Caesar class

Open the "Caesar.cs" file by double clicking on it in the Solution Explorer. To include the necessary namespaces in the class header, use the "using" statement followed by the name of the desired namespace. The CrypTool 2.0 API provides the following namespaces:

- Cryptool.PluginBase — contains interfaces such as IPlugin, IHash, and ISettings, as well as attributes, enumerations, delegates and extensions.

- Cryptool.PluginBase.Analysis — contains interfaces for cryptanalysis plugins (such as "Stream Comparator").

- Cryptool.PluginBase.Control — contains global interfaces for the IControl feature for defining custom controls.

- Cryptool.PluginBase.Cryptography — contains interfaces for encryption and hash algorithms such as AES, DES and MD5.

- Cryptool.PluginBase.Editor — contains interfaces for editors that can be implemented in Cryp-Tool 2.0, such as the default editor.

- Cryptool.PluginBase.Generator — contains interfaces for generators, including the random input generator.

- Cryptool.PluginBase.IO — contains interfaces for input, output and the CryptoolStream.

- Cryptool.PluginBase.Miscellaneous — contains assorted helper classes, including *GuiLogMessage* and *PropertyChanged*.

- Cryptool.PluginBase.Resources — used only by CrypWin and the editor; not necessary for plugin development.

- Cryptool.PluginBase.Tool — contains an interface for all external tools implemented by Cryp-Tool 2.0 that do not entirely support the CrypTool 2.0 API .

- Cryptool.PluginBase.Validation — contains interfaces for validation methods, including regular expressions.

In our example, the Caesar algorithm necessitates the inclusion of the following namespaces:

- Cryptool.PluginBase — to implement ISettings in the CaesarSettings class.

- Cryptool.PluginBase.Cryptography — to implement IEncryption in the Caesar class.

- Cryptool.PluginBase.IO — to use CryptoolStream for data input and output.

- Cryptool.PluginBase.Miscellaneous — to use the CrypTool event handler.

It is important to define a new default namespace for our public class ("Caesar"). In CrypTool 2.0 the standard namespace convention is *Cryptool.[name of class]*. Therefore our namespace will be defined as *Cryptool.Caesar*.

At this point, the source code should look like the following:

```
1  using System.Collections.Generic;
2  using System.Text;
3
4  //required CrypTool namespaces
5  using Cryptool.PluginBase;
6  using Cryptool.PluginBase.Cryptography;
7  using Cryptool.PluginBase.IO;
8  using Cryptool.PluginBase.Miscellaneous;
9
10 namespace Cryptool.Caesar
11 {
12   public class Caesar
13   {
14   }
15 }
```

Next we should let the "Caesar" class inherit from IEncryption by making the following alteration:

```
1  namespace Cryptool.Caesar
2  {
3    public class Caesar : IEncryption
4    {
5    }
6  }
```

2.4.4 Adding interface functions for the Caesar class

There is an underscore at the "I" in IEncryption statement. Move your mouse over it or place the cursor at it and press "Shift+Alt+F10" and you will see the following submenu:
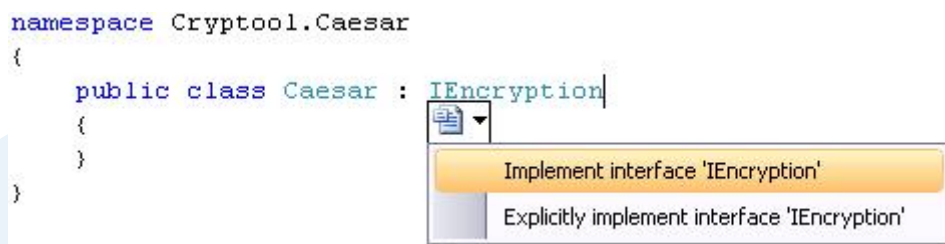
Figure 2.12: Inherit submenu

Choose the item "Implement interface 'IEncryption"'. Visual Studio/C# Express will now place all available and needed interface members to interact with the CrypTool core (this saves you also a lot of typing code).
Your code will now look like this:

```
1  using System.Collections.Generic;
2  using System.Text;
3
4  using Cryptool.PluginBase;
```

```
5  using Cryptool.PluginBase.Cryptography;
6  using Cryptool.PluginBase.IO;
7  using Cryptool.PluginBase.Miscellaneous;
8
9  namespace Cryptool.Caesar
10 {
11     public class Caesar : IEncryption
12     {
13         #region IPlugin Members
14
15         public void Dispose()
16         {
17             throw new NotImplementedException();
18         }
19
20         public void Execute()
21         {
22             throw new NotImplementedException();
23         }
24
25         public void Initialize()
26         {
27             throw new NotImplementedException();
28         }
29
30         public event GuiLogNotificationEventHandler
                OnGuiLogNotificationOccured;
31
32         public event PluginProgressChangedEventHandler
                OnPluginProgressChanged;
33
34         public event StatusChangedEventHandler OnPluginStatusChanged;
35
36         public void Pause()
37         {
38             throw new NotImplementedException();
39         }
40
41         public void PostExecution()
42         {
43             throw new NotImplementedException();
44         }
45
46         public void PreExecution()
47         {
48             throw new NotImplementedException();
49         }
50
51         public System.Windows.Controls.UserControl Presentation
52         {
```

```
53            get { throw new NotImplementedException(); }
54        }
55
56        public System.Windows.Controls.UserControl
             QuickWatchPresentation
57        {
58            get { throw new NotImplementedException(); }
59        }
60
61        public ISettings Settings
62        {
63            get { throw new NotImplementedException(); }
64        }
65
66        public void Stop()
67        {
68            throw new NotImplementedException();
69        }
70
71        #endregion
72
73        #region INotifyPropertyChanged Members
74
75        public event System.ComponentModel.PropertyChangedEventHandler
             PropertyChanged;
76
77        #endregion
78    }
79 }
```

2.4.5 Add namespace and interfaces for the class CaesarSettings

Let's now take a look at the second class "CaesarSettings" by double clicking at the "CaesarSettings.cs"
file at the Solution Explorer. First we also have to include the namespace of "Cryptool.PluginBase"
to the class header and let the settings class inherit from "ISettings" analogous as seen before at the
Caesar class. Visual Studio/C# Express will here also automatically place code from the CrypTool
interface if available.

```
1 using System.Collections.Generic;
2 using System.Text;
3
4 using Cryptool.PluginBase;
5
6 namespace Cryptool.Caesar
7 {
8     public class CaesarSettings : ISettings
9     {
10        #region ISettings Members
11
12        public bool HasChanges
```

```
13            {
14                get
15                {
16                    throw new NotImplementedException();
17                }
18                set
19                {
20                    throw new NotImplementedException();
21                }
22            }
23
24            #endregion
25
26            #region INotifyPropertyChanged Members
27
28            public event System.ComponentModel.PropertyChangedEventHandler
                   PropertyChanged;
29
30            #endregion
31        }
32 }
```

2.4.6 Add controls for the class CaesarSettings (if needed)

Now we have to implement some kind of controls (like button, text box) if we need them in the
CrypTool **TaskPane** to modify settings of the algorithm. If you decided to provide an algorithm
(e.g. Hash) which do not have any kind of settings you can leave this class now empty. The only
part you have to modify is the "HasChanges" property to avoid any "NotImplementedException".
How to modify this property you can see in the following code which demonstrate the modifications
fot the TaskPane for our Caesar algorithm. You can also take a look at the other algorithm source
codes which are stored in our subversion how you can provide a TaskPane. The following source code
demonstrates how we provide our TaskPane as seen above.

```
1 using System;
2 using System.ComponentModel;
3 using System.Windows;
4 using Cryptool.PluginBase;
5 using System.Windows.Controls;
6
7 namespace Cryptool.Caesar
8 {
9     public class CaesarSettings : ISettings
10    {
11        #region Public Caesar specific interface
12
13        /// <summary>
14        /// We use this delegate to send log messages from the
               settings class to the Caesar plugin
15        /// </summary>
16        public delegate void CaesarLogMessage(string msg,
```

```
              NotificationLevel loglevel);
17
18        /// <summary>
19        /// An enumaration for the different modes of dealing with
              unknown characters
20        /// </summary>
21        public enum UnknownSymbolHandlingMode { Ignore = 0, Remove =
              1, Replace = 2 };
22
23        /// <summary>
24        /// Fire if a new status message was send
25        /// </summary>
26        public event CaesarLogMessage LogMessage;
27
28        public delegate void CaesarReExecute();
29
30        public event CaesarReExecute ReExecute;
31
32        /// <summary>
33        /// Retrieves the current sihft value of Caesar (i.e. the key)
              , or sets it
34        /// </summary>
35        [PropertySaveOrder(0)]
36        public int ShiftKey
37        {
38            get { return shiftValue; }
39            set
40            {
41                setKeyByValue(value);
42            }
43        }
44
45        /// <summary>
46        /// Retrieves the current setting whether the alphabet should
              be treated as case sensitive or not
47        /// </summary>
48        [PropertySaveOrder(1)]
49        public bool CaseSensitiveAlphabet
50        {
51            get
52            {
53                if (caseSensitiveAlphabet == 0)
54                {   return false;   }
55                else
56                {   return true;    }
57            }
58            set {} // readonly, because there are some problems if we
                  omit the set part.
59        }
60
```

```
61
62          /// <summary>
63          /// Returns true if some settings have been changed. This
                value should be set externally to false e.g.
64          /// when a project was saved.
65          /// </summary>
66          [PropertySaveOrder(3)]
67          public bool HasChanges
68          {
69              get { return hasChanges; }
70              set { hasChanges = value; }
71          }
72
73          #endregion
74
75          #region Private variables
76          private bool hasChanges;
77          private int selectedAction = 0;
78          private string upperAlphabet = ''ABCDEFGHIJKLMNOPQRSTUVWXYZ'';
79          private string lowerAlphabet = ''abcdefghijklmnopqrstuvwxyz'';
80          private string alphabet = ''ABCDEFGHIJKLMNOPQRSTUVWXYZ'';
81          private char shiftChar = 'C';
82          private int shiftValue = 2;
83          // private int shiftValue = 2;
84          private UnknownSymbolHandlingMode unknownSymbolHandling =
                UnknownSymbolHandlingMode.Ignore;
85          private int caseSensitiveAlphabet = 0; // 0 = case insensitve,
                1 = case sensitive
86          private bool sensitivityEnabled = true;
87          #endregion
88
89          #region Private methods
90
91          private string removeEqualChars(string value)
92          {
93              int length = value.Length;
94
95              for (int i = 0; i < length; i++)
96              {
97                  for (int j = i + 1; j < length; j++)
98                  {
99                      if ((value[i] == value[j]) || (!
                          CaseSensitiveAlphabet & (char.ToUpper(value[i])
                           == char.ToUpper(value[j]))))
100                     {
101                         LogMessage(''Removing duplicate letter: \''' +
                                value[j] + ''\' from alphabet!'',
                                NotificationLevel.Warning);
102                         value = value.Remove(j,1);
103                         j--;
```

```
104                            length --;
105                        }
106                    }
107                }
108
109                return value;
110            }
111
112            /// <summary >
113            /// Set the new shiftValue and the new shiftCharacter to
                   offset % alphabet.Length
114            /// </summary >
115            private void setKeyByValue(int offset)
116            {
117                HasChanges = true;
118
119                // making sure the shift value lies within the alphabet
                       range
120                offset = offset % alphabet.Length;
121
122                // set the new shiftChar
123                shiftChar = alphabet[offset];
124
125                // set the new shiftValue
126                shiftValue = offset;
127
128                // Anounnce this to the settings pane
129                OnPropertyChanged(''ShiftValue'');
130                OnPropertyChanged(''ShiftChar'');
131
132                // print some info in the log.
133                LogMessage(''Accepted new shift value '' + offset + ''! (
                       Adjusted shift character to \''' + shiftChar + ''\')'',
                       NotificationLevel.Info);
134            }
135
136            private void setKeyByCharacter(string value)
137            {
138                try
139                {
140                    int offset;
141                    if (this.CaseSensitiveAlphabet)
142                    {
143                        offset = alphabet.IndexOf(value[0]);
144                    }
145                    else
146                    {
147                        offset = alphabet.ToUpper().IndexOf(char.ToUpper(
                               value[0]));
148                    }
```

```
149
150                if (offset >= 0)
151                {
152                    HasChanges = true;
153                    shiftValue = offset;
154                    shiftChar = alphabet[shiftValue];
155                    LogMessage(''Accepted new shift character \''' +
                            shiftChar + ''\'! (Adjusted shift value to '' +
                             shiftValue + '')'', NotificationLevel.Info);
156                    OnPropertyChanged(''ShiftValue'');
157                    OnPropertyChanged(''ShiftChar'');
158                }
159                else
160                {
161                    LogMessage(''Bad input \'''' + value + ''\''! (
                            Character not in alphabet!) Reverting to '' +
                            shiftChar.ToString() + ''!'', NotificationLevel
                            .Error);
162                }
163            }
164            catch (Exception e)
165            {
166                LogMessage(''Bad input \'''' + value + ''\''! ('' + e.
                        Message + '') Reverting to '' + shiftChar.ToString
                        () + ''!'', NotificationLevel.Error);
167            }
168        }
169
170        #endregion
171
172        #region Algorithm settings properties (visible in the Settings
            pane)
173
174        [PropertySaveOrder(4)]
175        [ContextMenu(''Action'', ''Select the Algorithm action'', 1,
            DisplayLevel.Beginner, ContextMenuControlType.ComboBox, new
            int[] { 1, 2 }, ''Encrypt'', ''Decrypt'')]
176        [TaskPane(''Action'', ''setAlgorithmActionDescription'', null,
            1, true, DisplayLevel.Beginner, ControlType.ComboBox, new
            string[] { ''Encrypt'', ''Decrypt'' })]
177        public int Action
178        {
179            get
180            {
181                return this.selectedAction;
182            }
183            set
184            {
185                if (value != selectedAction) HasChanges = true;
186                this.selectedAction = value;
```

```
187                     OnPropertyChanged(''Action'');
188
189                     if (ReExecute != null) ReExecute();
190             }
191         }
192
193         [PropertySaveOrder(5)]
194         [TaskPane(''Key as integer'', ''Enter the number of letters to
               shift. For instance a value of 1 means that the plaintext
               character a gets mapped to the ciphertext character B, b to
                C and so on.'', null, 2, true, DisplayLevel.Beginner,
               ControlType.NumericUpDown, ValidationType.RangeInteger, 0,
               100)]
195         public int ShiftValue
196         {
197             get { return shiftValue; }
198             set
199             {
200                 setKeyByValue(value);
201                 if (ReExecute != null) ReExecute();
202             }
203         }
204
205
206         [PropertySaveOrder(6)]
207         [TaskPaneAttribute(''Key as single letter'', ''Enter a single
               letter as the key. This letter is mapped to an integer
               stating the position in the alphabet. The values for 'Key
               as integer' and 'Key as single letter' are always
               synchronized.'', null, 3, true, DisplayLevel.Beginner,
               ControlType.TextBox, ValidationType.RegEx, ''^([A-Z]|[a-z])
               {1,1}'')]
208         public string ShiftChar
209         {
210             get { return this.shiftChar.ToString(); }
211             set
212             {
213                 setKeyByCharacter(value);
214                 if (ReExecute != null) ReExecute();
215             }
216         }
217
218         [PropertySaveOrder(7)]
219         [ContextMenu(''Unknown symbol handling'', ''What should be
               done with encountered characters at the input which are not
                in the alphabet?'', 4, DisplayLevel.Expert,
               ContextMenuControlType.ComboBox, null, new string[] { ''
               Ignore (leave unmodified)'', ''Remove'', ''Replace with
               \'?\''' })]
220         [TaskPane(''Unknown symbol handling'', ''What should be done
```

```csharp
                with encountered characters at the input which are not in
                the alphabet?'', null, 4, true, DisplayLevel.Expert,
                ControlType.ComboBox, new string[] { ''Ignore (leave
                unmodified)'', ''Remove'', ''Replace with \'?\''' })]
221         public int UnknownSymbolHandling
222         {
223             get { return (int)this.unknownSymbolHandling; }
224             set
225             {
226                 if ((UnknownSymbolHandlingMode)value !=
                        unknownSymbolHandling) HasChanges = true;
227                 this.unknownSymbolHandling = (
                        UnknownSymbolHandlingMode)value;
228                 OnPropertyChanged(''UnknownSymbolHandling'');
229
230                 if (ReExecute != null) ReExecute();
231             }
232         }
233
234         [SettingsFormat(0, ''Normal'', ''Normal'', ''Black'', ''White
                '', Orientation.Vertical)]
235         [PropertySaveOrder(9)]
236         [TaskPane(''Alphabet'', ''This is the used alphabet.'', null,
                6, true, DisplayLevel.Expert, ControlType.TextBox, '''')]
237         public string AlphabetSymbols
238         {
239           get { return this.alphabet; }
240           set
241           {
242             string a = removeEqualChars(value);
243             if (a.Length == 0) // cannot accept empty alphabets
244             {
245               LogMessage(''Ignoring empty alphabet from user! Using
                      previous alphabet: \''' + alphabet + ''\'' ('' +
                      alphabet.Length.ToString() + '' Symbols)'',
                      NotificationLevel.Info);
246             }
247             else if (!alphabet.Equals(a))
248             {
249               HasChanges = true;
250               this.alphabet = a;
251               setKeyByValue(shiftValue); //re-evaluate if the
                      shiftvalue is still within the range
252               LogMessage(''Accepted new alphabet from user: \''' +
                      alphabet + ''\'' ('' + alphabet.Length.ToString() +
                      '' Symbols)'', NotificationLevel.Info);
253               OnPropertyChanged(''AlphabetSymbols'');
254
255               if (ReExecute != null) ReExecute();
256             }
```

```
257                }
258            }
259
260        /// <summary>
261        /// Visible setting how to deal with alphabet case. 0 = case
               insentive , 1 = case sensitive
262        /// </summary>
263        //[ SettingsFormat (1, ''Normal '')]
264        [PropertySaveOrder (8)]
265        [ContextMenu (''Alphabet case sensitivity '', ''Should upper and
                lower case be treated differently? (Should a == A)'', 7,
             DisplayLevel.Expert , ContextMenuControlType.ComboBox , null,
              new string[] { ''Case insensitive '', ''Case sensitive '' })
              ]
266        [TaskPane (''Alphabet case sensitivity '', ''Should upper and
               lower case be treated differently? (Should a == A)'', null,
               7, true, DisplayLevel.Expert , ControlType.ComboBox , new
               string[] { ''Case insensitive '', ''Case sensitive '' })]
267        public int AlphabetCase
268        {
269            get { return this.caseSensitiveAlphabet; }
270            set
271            {
272                if (value != caseSensitiveAlphabet) HasChanges = true;
273                this.caseSensitiveAlphabet = value;
274                if (value == 0)
275                {
276                    if (alphabet == (upperAlphabet + lowerAlphabet))
277                    {
278                        alphabet = upperAlphabet;
279                        LogMessage (''Changing alphabet to: \'''' +
                            alphabet + ''\'' ('' + alphabet.Length.
                            ToString () + '' Symbols)'',
                            NotificationLevel.Info);
280                        OnPropertyChanged (''AlphabetSymbols '');
281                        // re-set also the key (shiftvalue/shiftChar
                            to be in the range of the new alphabet
282                        setKeyByValue (shiftValue);
283                    }
284                }
285                else
286                {
287                    if (alphabet == upperAlphabet)
288                    {
289                        alphabet = upperAlphabet + lowerAlphabet;
290                        LogMessage (''Changing alphabet to: \'''' +
                            alphabet + ''\'' ('' + alphabet.Length.
                            ToString () + '' Symbols)'',
                            NotificationLevel.Info);
291                        OnPropertyChanged (''AlphabetSymbols '');
```

```
292                       }
293                   }
294
295                   // remove equal characters from the current alphabet
296                   string a = alphabet;
297                   alphabet = removeEqualChars(alphabet);
298
299                   if (a != alphabet)
300                   {
301                       OnPropertyChanged(''AlphabetSymbols'');
302                       LogMessage(''Changing alphabet to: \'''' +
                              alphabet + ''\'' ('' + alphabet.Length.ToString
                              () + '' Symbols)'', NotificationLevel.Info);
303                   }
304
305                   OnPropertyChanged(''AlphabetCase'');
306                   if (ReExecute != null) ReExecute();
307               }
308           }
309
310           #endregion
311
312           #region INotifyPropertyChanged Members
313
314           public event PropertyChangedEventHandler PropertyChanged;
315
316           protected void OnPropertyChanged(string name)
317           {
318             if (PropertyChanged != null)
319             {
320               PropertyChanged(this, new PropertyChangedEventArgs(name));
321             }
322           }
323
324           #endregion
325
326           #region TaskPaneAttributeChanged-Sample
327           /// <summary>
328           /// This event is just used here for sample reasons
329           /// </summary>
330           public event TaskPaneAttributeChangedHandler
                  TaskPaneAttributeChanged;
331
332           [TaskPane(''Enable/Disable sensitivity'', ''This setting is
                  just a sample and shows how to enable / disable a setting
                  .'', ''AttributeChangedSample'', 8, false, DisplayLevel.
                  Beginner, ControlType.Button)]
333           public void EnableDisableSesitivity()
334           {
335             if (TaskPaneAttributeChanged!= null)
```

```
336              {
337                sensitivityEnabled = !sensitivityEnabled;
338                if (sensitivityEnabled)
339                {
340                  TaskPaneAttributeChanged(this, new
                         TaskPaneAttributeChangedEventArgs(new
                         TaskPaneAttribteContainer(''AlphabetCase'',
                         Visibility.Visible)));
341                }
342                else
343                {
344                  TaskPaneAttributeChanged(this, new
                         TaskPaneAttributeChangedEventArgs(new
                         TaskPaneAttribteContainer(''AlphabetCase'',
                         Visibility.Collapsed)));
345                }
346              }
347            }
348            #endregion TaskPaneAttributeChanged-Sample
349        }
350 }
```

2.5 Select and add an image as icon for the class Caesar

Before we go back to the code of the Caesar class, we have to add an icon image to our project, which will be shown in the CrypTool **ribbon bar** or/and **navigation pane**. As there is no default, using an icon image is mandatory.

Note: This will be changed in future. A default icon will be used if no icon image has been provided.

For testing purposes you may create a simple black and white PNG image with MS Paint or Paint.NET. As image size you can use 40x40 pixels for example, but as the image will be scaled when required, any size should do it. Place the image file in your project directory or in a subdirectory.

Then make a right click on the project item "Caesar" or any subdirectory within the Solution Explorer, and select "Add->Existing Item...":
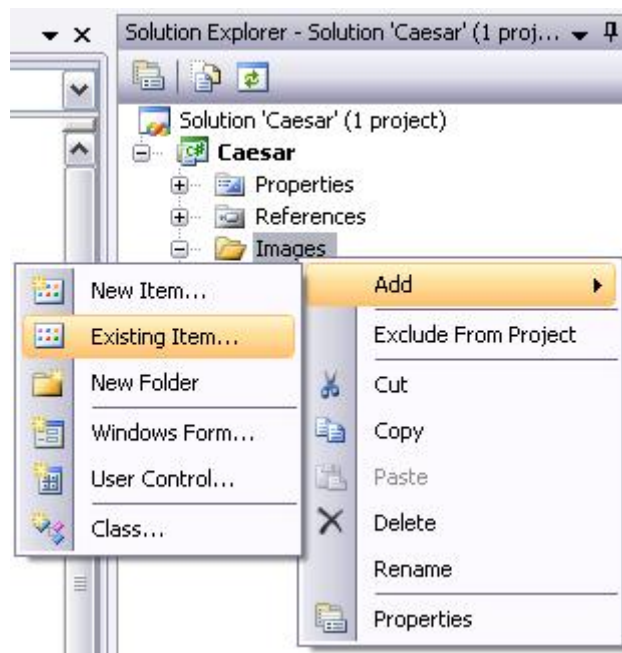
Figure 2.13: Add existing item

As you can see, in our solution we create an new folder named "Images" (make a right click on the project item "Caesar" and select "Add->New Folder") and placed there the new icon by clicking right on the folder as mentioned aboved.

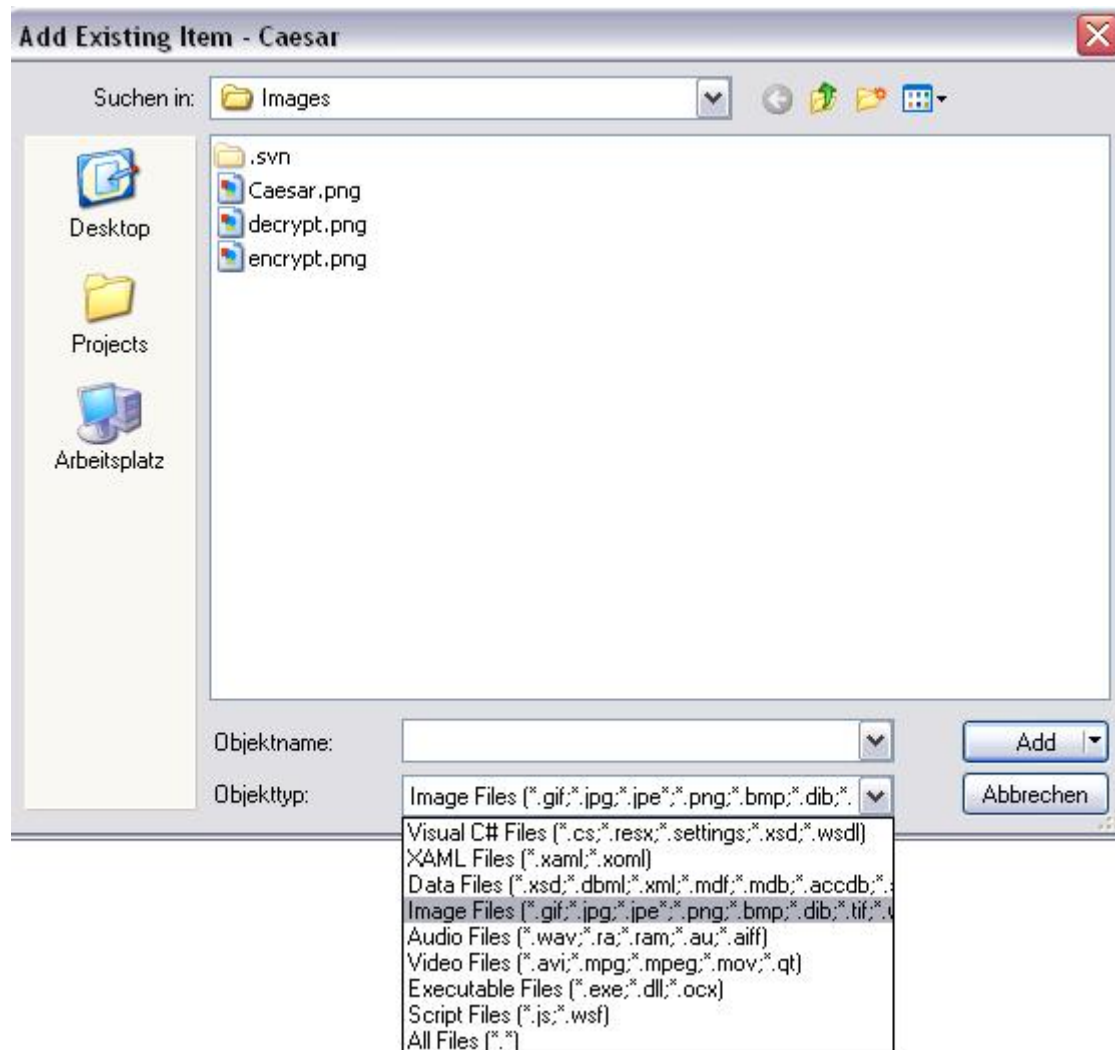Then select "Image Files" as file type, and choose the icon for your plugin: Finally we have to set

Figure 2.14: Choose the right icon

the icon as a "Resource" to avoid providing the icon as a separate file. Make a right click on the icon and select the item "Properties":
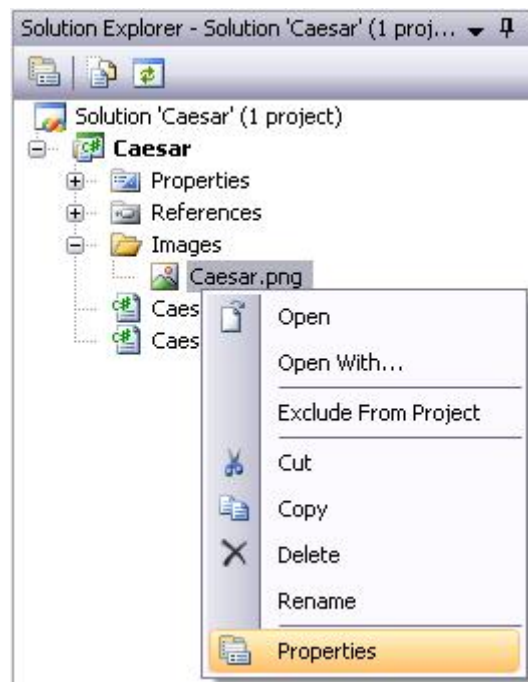
Figure 2.15: Icon properties

In the "Properties" panel you have to set the "Build Action" to "Resource" (not embedded resource):
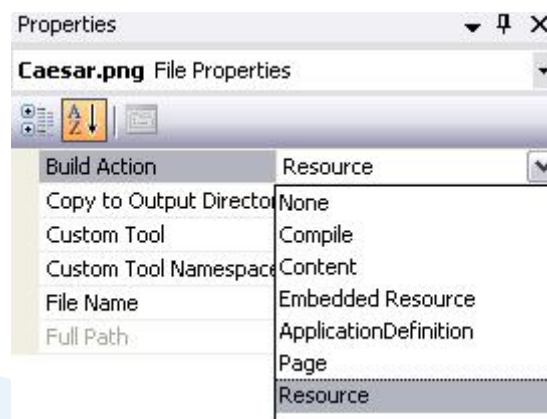
Figure 2.16: Icon build action

2.6 Set the attributes for the class Caesar

Now let's go back to the code of the Caesar class ("Caesar.cs" file). First we have to set the necessary attributes for our class. This attributes are used to provide additional information for the CrypTool 2.0 environment. If not set, your plugin won't show up in the GUI, even if everything else is implemented correctly.

Attributes are used for **declarative** programming and provide meta data, that can be attached to the existing .NET meta data , like classes and properties. CrypTool provides a set of custom attributes, that are used to mark the different parts of your plugin.

[Author]

The first attribute called "Author" is optional, which means we are not forced to define this attribute. It provides the additional information about the plugin developer. This informations you can see for example in the TaskPane as shown on a screenshot above. We set this attribute to demonstrate how it has to look in case you want to provide this attribute.

```
[Author(
  1 of 2   AuthorAttribute.AuthorAttribute (string author, string email, string institute, string url)
{
```
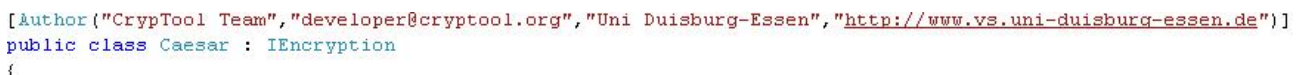
Figure 2.17: Attribute author

As we can see above the author attribute takes four elements of type string. These elements are:

- Author = name of the plugin developer

- Email = email of the plugin developer if he wants to be contact

- Institute = current employment of the developer like University or Company

- Url = the website or homepage of the developer

All this elements are also optional. The developer decides what he wants to publish. Unused elements shall be set to null or a zero-length string (""").

Our author attribute should look now as you can see below:
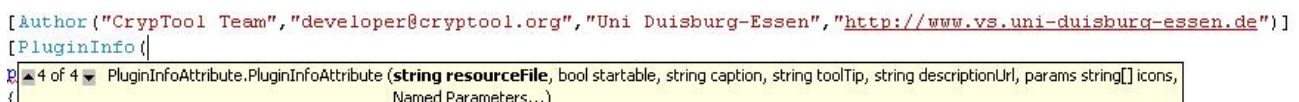
```
[Author("CrypTool Team","developer@cryptool.org","Uni Duisburg-Essen","http://www.vs.uni-duisburg-essen.de")]
public class Caesar : IEncryption
{
```

Figure 2.18: Filled author auttribute

[PluginInfo]

The second attribute called "PluginInfo" provides the necessary information about the plugin like caption and tool tip. This attribute is mandatory. The attribute has the definition as you can see below:

```
[Author("CrypTool Team","developer@cryptool.org","Uni Duisburg-Essen","http://www.vs.uni-duisburg-essen.de")]
[PluginInfo(
  4 of 4   PluginInfoAttribute.PluginInfoAttribute (string resourceFile, bool startable, string caption, string toolTip, string descriptionUrl, params string[] icons,
{                                  Named Parameters...)
```

Figure 2.19: Attribute PluginInfo

This attribute expects the following elements:

- resourceFile = Defines if resource files will be provided and where to find them. E.g. to provide the plugin multilingual you can store the labels in such a resource file. This element is optional.

- startable = Set this flag to true only if your plugin is some kind of input or generator plugin (probably if your plugin just has outputs and no inputs). In all other cases use false here. This flag is important. Setting this flag to true for a non input/generator plugin will result in unpredictable chain runs. This element is mandatory.

- caption = from type string, the name of the plugin or the resource field name if you provide the caption in a resource file (e.g. to provide the button content). This element is mandatory.

- toolTip = from type string, description of the plugin or the resource field name if you provide the toolTip in a resource file (e.g. to provide the button tool tip). This element is optional.

- descriptionUrl = from type string, define where to find the whole description files (e.g. XAML files). This element is optional.

- icons = from type string array, which provides all necessary icon paths you want to use in the plugin (e.g. the plugin icon as seen above). This element is mandatory.

Unused optional elements shall be set to null or a zero-length string ("").

Note 1: It is possible to use the plugin without setting a caption though it is not recommended. This will be changed in future and the plugin will fail to load without a caption.

Note 2: Currently a zero-length toolTip string appears as empty box. This will be changed in future.

Note 3: Tooltip and description currently do not support internationalization and localization. This will be changed in future.

In our example the first parameter called "resourceFile" has to be set to "Cryptool.Caesar.Resource.res" because we want to provide the plugin multilingual and want to store the labels and caption in a resource file. Otherwise ignore this element.

```
[PluginInfo("Cryptool.Caesar.Resources.res",
p ▲ 4 of 4 ▼  PluginInfoAttribute.PluginInfoAttribute (string resourceFile, bool startable, string caption, string toolTip, string descriptionUrl, params string[] icons,
{                                          Named Parameters...)
```

Figure 2.20: Attribute PluginInfo element resourceFile

The second parameter called "startable" has to be set to "false", because our encryption algorithm is neither an input nor generator plugin.

```
[PluginInfo("Cryptool.Caesar.Resources.res",false,
p ▲ 4 of 4 ▼  PluginInfoAttribute.PluginInfoAttribute (string resourceFile, bool startable, string caption, string toolTip, string descriptionUrl, params string[] icons,
{                                          Named Parameters...)
```

Figure 2.21: Attribute PluginInfo startable

The next two parameters are needed to define the plugin's name and its description. Now that we decided to provide a resource file we have to place here the both resource field names which contains the description and captions. Otherwise just write here a simple string text:

```
[PluginInfo("Cryptool.Caesar.Resources.res",false, "pluginName", "pluginToolTip",
p ▲ 4 of 4 ▼  PluginInfoAttribute.PluginInfoAttribute (string resourceFile, bool startable, string caption, string toolTip, string descriptionUrl, params string[] icons,
{                                          Named Parameters...)
```

Figure 2.22: Attribute PluginInfo name and description

The next element defines the location path of the description file. The parameter is made up by <Assembly name>/<filename> or <Assembly name>/<Path>/<file name> if you want to store your description files in a separate folder (as seen on the icon). The description file has to be of type XAML. In our case we create a folder called "DetailedDescription" and store our XAML file there with the necessary images if needed. How you manage the files and folders is up to you. This folder could now look as you can see below:

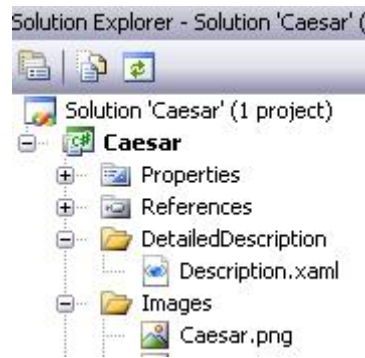Accordingly the attribute parameter has to be set to:

Figure 2.23: Attribute PluginInfo icon and description file path

Figure 2.24: Attribute PluginInfo description file

The detailed description could now look like this in CrypTool (right click plugin icon on workspace and select "Show description"):
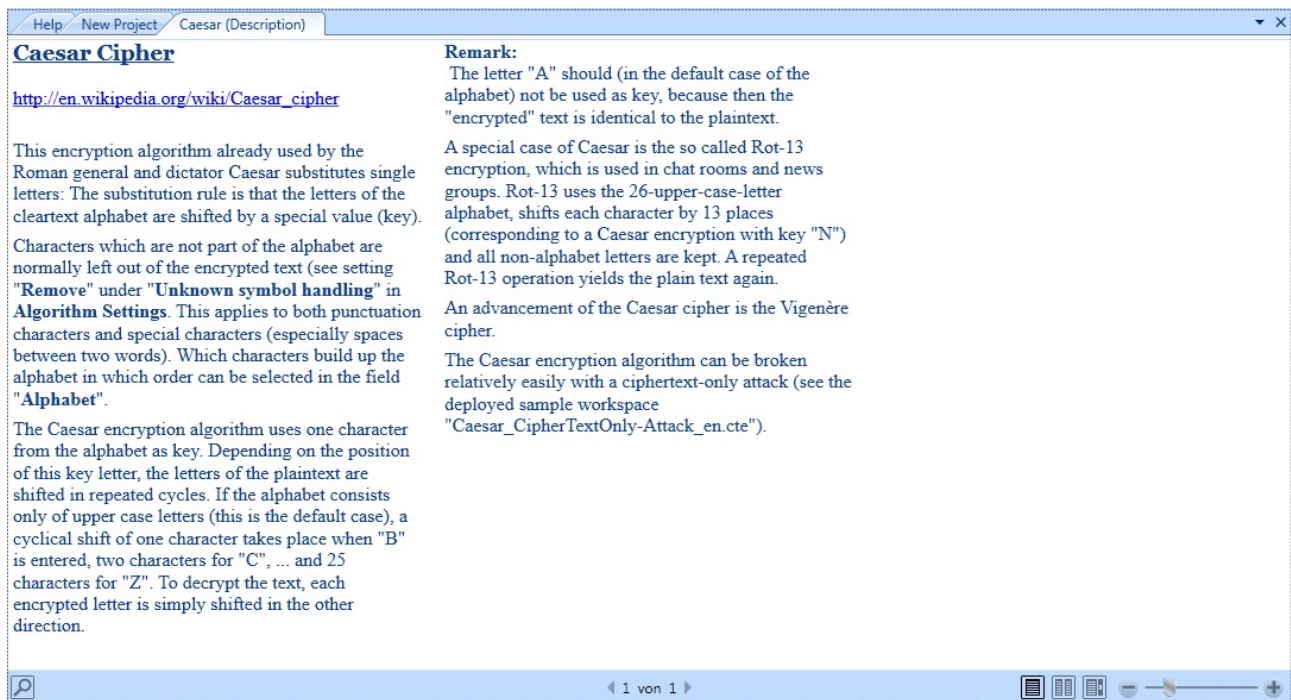
Figure 2.25: XAML detailed description

The last parameter tells CrypTool the names of the provided icons. This parameter is made up by <Assembly name>/<file name> or <Assembly name>/<Path>/<file name>.

The most important icon is the plugin icon, which will be shown in CrypTool in the ribbon bar or navigation pane (This is the first icon in list, so you have to provide at least one icon for a plugin). As named above how to add an icon to the solution accordingly we have to tell CrypTool where to find the icon by setting this parameter as you can see below:

```
[PluginInfo("Cryptool.Caesar.Resources.res",false, "pluginName", "pluginToolTip", "Caesar/DetailedDescription/Description.xaml",
    "Caesar/Images/Caesar.png", "Caesar/Images/encrypt.png", "Caesar/Images/decrypt.png")]
public class Caesar : IEncryption
```

Figure 2.26: Attribute PluginInfo icons

You can define further icon paths if needed, by adding the path string separated by a comma. We just add here two further icons (don't forget to add the icons to your solution) to provide them for the context menu in the CrypTool workspace.

[EncryptionType]
The third and last attribute called "EncryptionType" is needed to tell CrypTool which type of plugin we want to provide. CrypTool is now able to place the plugin in the right group at the navigation pane or/and ribbon bar. Therefore Caesar is a classical algorithm so we have to set the following attribute:

```
[Author("CrypTool Team","developer@cryptool.org","Uni Duisburg-Essen","http://www.vs.uni-duisburg-essen.de")]
[PluginInfo("Cryptool.Caesar.Resources.res",false, "pluginName", "pluginToolTip", "Caesar/DetailedDescription/Description.xaml",
    "Caesar/Images/Caesar.png", "Caesar/Images/encrypt.png", "Caesar/Images/decrypt.png")]
[EncryptionType(EncryptionType.Classic)]
public class Caesar : IEncryption
{
```

Figure 2.27: Attribute encryption type

The "EncryptionType" attribute can also be set as the following types:

- Asymmetric = for asymmetric encryption algorithms like RSA

- Classic = for classic encryption or hash algorithms like Caesar or MD5

- Hybrid = for a combination of several algorithm where the data is encrypted symmetric and the encryption key asymmetric

- SymmetricBlock = for all block cipher algorithms like DES, AES or Twofish

- SymmetricStream = for all stream cipher algorithms like RC4, Rabbit or SEAL

2.7 Set the private variables for the settings in the class Caesar

The next step is to define some private variables needed for the settings, input and output data which could look like this:

```
1 public class Caesar : IEncryption
2 {
3   #region Private variables
4   private CaesarSettings settings;
5   private string inputString;
6   private string outputString;
7   private enum CaesarMode { encrypt, decrypt };
8   private List<CryptoolStream> listCryptoolStreamsOut = new List<
      CryptoolStream>();
9   #endregion
```

Please notice if there is a sinuous line at the code you type for example at the "CryptoolStream" type of the variable listCryptoolStreamsOut. "CryptoolStream" is a data type for input and output between plugins and is able to handle large data amounts. To use the CrypTool own stream type, include the namespace "Cryptool.PluginBase.IO" with a "using" statement as explained in chapter **??**. Check the other code entries while typing and update the missing namespaces.

The following private variables are being used in this example:

- CaesarSettings settings: required to implement the IPlugin interface properly

- string inputString: sting to read the input data from

- string outputString: string to save the output data

- enum CaesarMode: our own definition how to select between an encryption or decryption. It's up to you how to solve your algorithm

- List<CryptoolStream> listCryptoolStreamsOut: list of all streams being created by Caesar plugin, required to perform a clean dispose

2.8 Define the code of the class Caesar to fit the interface

Next we have to complete our code to correctly serve the interface.
First we add a constructor to our class where we can create an instance of our settings class and a function to handle events:

```
1  public class Caesar : IEncryption
2  {
3    #region Private variables
4    private CaesarSettings settings;
5    private string inputString;
6    private string outputString;
7    private enum CaesarMode { encrypt, decrypt };
8    private List<CryptoolStream> listCryptoolStreamsOut = new List<
         CryptoolStream >();
9    #endregion
10
11   public Caesar()
12   {
13     this.settings = new CaesarSettings();
14     this.settings.LogMessage += Caesar_LogMessage;
15   }
```

Secondly, we have to implement the property "Settings" defined in the interface:

```
1  public ISettings Settings
2  {
3    get { return (ISettings)this.settings; }
4    set { this.settings = (CaesarSettings)value; }
5  }
```

Thirdly we have to define five properties with their according attributes. This step is necessary to tell CrypTool that these properties are input/output properties used for data exchange with other plugins or to provide our plugin with external data.

The attribute is named "PropertyInfo" and consists of the following elements:

- direction = defines whether this property is an input or output property, i.e. whether it reads input data or writes output data
  - Direction.Input
  - Direction.Output

- caption = caption of the property (e.g. shown at the input on the dropped icon in the editor), see below:
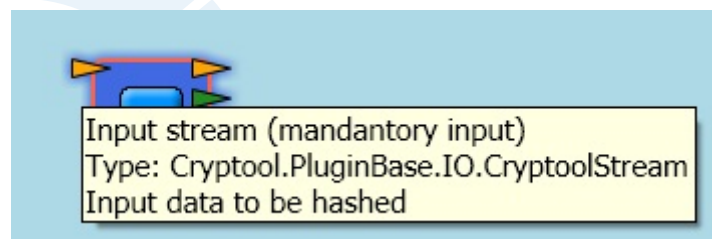
Figure 2.28: Possible property caption

- toolTip = tooltip of the property (e.g. shown at the input arrow on the dropped icon in the editor), see above

- descriptionUrl = not used right now

- mandatory = this flag defines whether an input is required to be connected by the user. If set to true, there has to be an input connection that provides data. If no input data is provided for mandatory input, your plugin will not be executed in the workflow chain. If set to false, connecting the input is optional. This only applies to input properties. If using Direction.Output, this flag is ignored.

- hasDefaultValue = if this flag is set to true, CrypTool treats this plugin as though the input has already input data.

- DisplayLevel = define in which display levels your property will be shown in CrypTool. CrypTool provides the following display levels:

  - DisplayLevel.Beginner

  - DisplayLevel.Experienced

  - DisplayLevel.Expert

  - DisplayLevel.Professional

- QuickWatchFormat = defines how the content of the property will be shown in the quick watch. CrypTool accepts the following quick watch formats:

  - QuickWatchFormat.Base64

  - QuickWatchFormat.Hex

  - QuickWatchFormat.None

  - QuickWatchFormat.Text
    A quick watch in Hex could look like this:
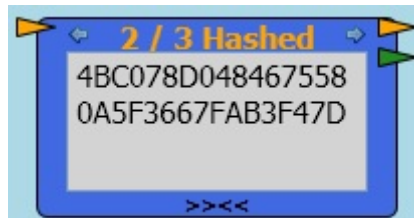
Figure 2.29: Possible quick watch

- quickWatchConversionMethod = this string points to a conversion method; most plugins can use a "null" value here, because no conversion is necessary. The QuickWatch function uses system "default" encoding to display data. So only if your data is in some other format, like Unicode or UTF8, you have to provide the name of a conversion method as string. The method header has to look like this:

```
1  object YourMethodName(string PropertyNameToConvert)
```

First we define the "InputString" property getter and setter which is needed to provide our plugin with data which has to be encrypted or decrypted:

```
1  [PropertyInfo(Direction.InputData, ''Text input'', ''Input a string to
      be processed by the Caesar cipher'', '''', true, false,
    DisplayLevel.Beginner, QuickWatchFormat.Text, null)]
```

```
2 public string InputString
3 {
4   get { return this.inputString; }
5   set
6   {
7     if (value != inputString)
8     {
9       this.inputString = value;
10      OnPropertyChanged(''InputString'');
11    }
12  }
13 }
```

In the getter we return the value of the input data.

Note 1: It is currently not possible to read directly from the input data stream without creating an intermediate CryptoolStream.

Note 2: The naming may be confusing. The new CryptoolStream is not an output stream, but it is added to the list of output streams to enable a clean dispose afterwards. See chapter 9 below.

The setter checkes if the input value has changed and sets the new input data and announces the data to the CrypTool 2.0 environment by using the expression "OnPropertyChanged(<Property name>)". For input properties this step is necessary to update the quick watch view.
The output data property (which provides the encrypted or decrypted input data) could look like this:

```
1 [PropertyInfo(Direction.OutputData, ''Text output'', ''The string
    after processing with the Caesar cipher'', '''', false, false,
    DisplayLevel.Beginner, QuickWatchFormat.Text, null)]
2 public string OutputString
3 {
4   get { return this.outputString; }
5   set
6   {
7     outputString = value;
8     OnPropertyChanged(''OutputString'');
9   }
10 }
```

CrypTool does not require implementing output setters, as they will never be called from outside of the plugin. Nevertheless in this example our plugin accesses the property itself, therefore we chose to implement the setter.
You can also provide additional output data types if you like. For example we provide also an output data of type CryptoolStream, an input data for external alphabets and an input data for the shift value of our Caesar algorithm:

```
1 [PropertyInfo(Direction.OutputData, ''propStreamOutputToolTip'', ''
    propStreamOutputDescription'', '''', false, false, DisplayLevel.
    Beginner, QuickWatchFormat.Text, null)]
2 public CryptoolStream OutputData
3 {
4   get
```

```
5    {
6      if (outputString != null)
7      {
8        CryptoolStream cs = new CryptoolStream();
9        listCryptoolStreamsOut.Add(cs);
10       cs.OpenRead(Encoding.Default.GetBytes(outputString.ToCharArray()
            ));
11       return cs;
12     }
13     else
14     {
15       return null;
16     }
17   }
18   set { }
19 }
20
21 [PropertyInfo(Direction.InputData, ''External alphabet input'', ''
      Input a string containing the alphabet which should be used by
      Caesar.\nIf no alphabet is provided on this input, the internal
      alphabet will be used.'', '''', false, false, DisplayLevel.Expert,
      QuickWatchFormat.Text, null)]
22 public string InputAlphabet
23 {
24   get { return ((CaesarSettings)this.settings).AlphabetSymbols; }
25   set
26   {
27     if (value != null && value != settings.AlphabetSymbols)
28     {
29       ((CaesarSettings)this.settings).AlphabetSymbols = value;
30       OnPropertyChanged(''InputAlphabet'');
31     }
32   }
33 }
34
35 [PropertyInfo(Direction.InputData, ''Shift value (integer)'', ''Same
      setting as Shift value in Settings-Pane but as dynamic input.'',
      '''', false, false, DisplayLevel.Expert, QuickWatchFormat.Text,
      null)]
36 public int ShiftKey
37 {
38   get { return settings.ShiftKey; }
39   set
40   {
41     if (value != settings.ShiftKey)
42     {
43       settings.ShiftKey = value;
44     }
45   }
46 }
```

This property's setter is not called and therefore not implemented.

The CrypTool-API provides two methods to send messages to the CrypTool. The method "GuiLogMessage" is used to send messages to the CrypTool status bar. This is a nice feature to inform the user what your plugin is currently doing.
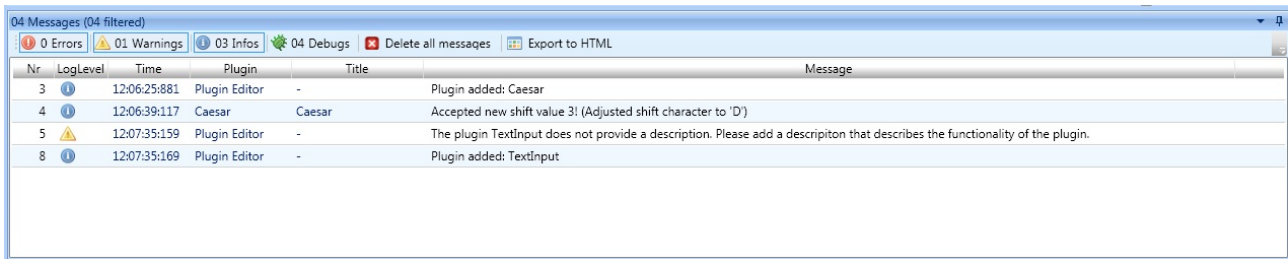

Figure 2.30: Status Bar

The method takes two parameters which are:

- Message = will be shown in the status bar and is of type string

- NotificationLevel = to group the messages to their alert level
  - NotificationLevel.Error
  - NotificationLevel.Warning
  - NotificationLevel.Info
  - NotificationLevel.Debug

As we can recognize we have two methods named "OnPropertyChanged" and "GuiLogMessage" which are not defined. So we have to define these two methods as you can see below:

```
public event GuiLogNotificationEventHandler
    OnGuiLogNotificationOccured;
private void GuiLogMessage(string message, NotificationLevel logLevel)
{
  EventsHelper.GuiLogMessage(OnGuiLogNotificationOccured, this, new
      GuiLogEventArgs(message, this, logLevel));
}

public event PropertyChangedEventHandler PropertyChanged;

public void OnPropertyChanged(String name)
{
  EventsHelper.PropertyChanged(PropertyChanged, this, new
      PropertyChangedEventArgs(name));
}
```

To use the "PropertyChangedEventHandler" you have to include the namespace "System.ComponentModel". Our whole included namespaces looks now like this:

```
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Windows.Control;

```

```
6 using Cryptool.PluginBase;
7 using Cryptool.PluginBase.Cryptography;
8 using Cryptool.PluginBase.IO;
9 using Cryptool.PluginBase.Miscellaneous;
```

2.9 Complete the actual code for the class Caesar

Up to now, the plugin is ready for the CrypTool base application to be accepted and been shown correctly in the CrypTool menu. What we need now, is the implementation of the actual algorithm in the function "Execute()" which is up to you as the plugin developer. CrypTool will always call first the Execute() function.If you place the whole algorithm in this function or split in other as needed is also up to you.

We decided to split our algorithm encryption and decryption in two separate functions, which finally call the function ProcessCaesar.

Let us demonstrate the Execute() function, too:

```
1 private void ProcessCaesar(CaesarMode mode)
2 {
3   CaesarSettings cfg = (CaesarSettings)this.settings;
4   StringBuilder output = new StringBuilder('''');
5   string alphabet = cfg.AlphabetSymbols;
6
7   // in case we want don't consider case in the alphabet, we use only
        capital letters, hence transform
8   // the whole alphabet to uppercase
9   if (!cfg.CaseSensitiveAlphabet)
10  {
11    alphabet = cfg.AlphabetSymbols.ToUpper(); ;
12  }
13
14  if (inputString != null)
15  {
16    for (int i = 0; i < inputString.Length; i++)
17    {
18      // get plaintext char which is currently processed
19      char currentchar = inputString[i];
20
21      // remember if it is upper case (otherwise lowercase is assumed)
22      bool uppercase = char.IsUpper(currentchar);
23
24      // get the position of the plaintext character in the alphabet
25      int ppos = 0;
26      if (cfg.CaseSensitiveAlphabet)
27      {
28        ppos = alphabet.IndexOf(currentchar);
29      }
30      else
31      {
32        ppos = alphabet.IndexOf(char.ToUpper(currentchar));
33      }
```

```
34
35      if (ppos >= 0)
36      {
37        // we found the plaintext character in the alphabet, hence we
            do the shifting
38        int cpos = 0; ;
39        switch (mode)
40        {
41          case CaesarMode.encrypt:
42            cpos = (ppos + cfg.ShiftKey) % alphabet.Length;
43            break;
44          case CaesarMode.decrypt:
45            cpos = (ppos - cfg.ShiftKey + alphabet.Length) % alphabet.
                Length;
46            break;
47        }
48
49        // we have the position of the ciphertext character, hence
            just output it in the correct case
50        if (cfg.CaseSensitiveAlphabet)
51        {
52          output.Append(alphabet[cpos]);
53        }
54        else
55        {
56          if (uppercase)
57          {
58            output.Append(char.ToUpper(alphabet[cpos]));
59          }
60          else
61          {
62            output.Append(char.ToLower(alphabet[cpos]));
63          }
64        }
65      }
66      else
67      {
68        // the plaintext character was not found in the alphabet,
            hence proceed with handling unknown characters
69        switch ((CaesarSettings.UnknownSymbolHandlingMode)cfg.
            UnknownSymbolHandling)
70        {
71          case CaesarSettings.UnknownSymbolHandlingMode.Ignore:
72            output.Append(inputString[i]);
73            break;
74          case CaesarSettings.UnknownSymbolHandlingMode.Replace:
75            output.Append('?');
76            break;
77        }
78      }
```

```
79
80        //show the progress
81        if (OnPluginProgressChanged != null)
82        {
83          OnPluginProgressChanged(this, new PluginProgressEventArgs(i,
                inputString.Length - 1));
84        }
85      }
86      outputString = output.ToString();
87      OnPropertyChanged(''OutputString'');
88      OnPropertyChanged(''OutputData'');
89    }
90 }
91
92 public void Encrypt()
93 {
94   ProcessCaesar(CaesarMode.encrypt);
95 }
96
97 public void Decrypt()
98 {
99   ProcessCaesar(CaesarMode.decrypt);
100 }
101
102 public void Execute()
103 {
104   switch (settings.Action)
105   {
106     case 0:
107       Caesar_LogMessage(''encrypting'', NotificationLevel.Debug);
108       Encrypt();
109       break;
110     case 1:
111       Decrypt();
112       break;
113     default:
114       break;
115   }
116 }
```

It is important to make sure that all changes of output properties will be announced to the Cryp-Tool environment. In this example this happens by calling the setter of OutputData which in turn calls "OnPropertyChanged" for both output properties "OutputData" and "OutputDataStream". Instead of calling the property's setter you can as well call "OnPropertyChanged" directly within the "Execute()" method.

Certainly you have seen the unknown method "ProgressChanged" which you can use to show the current algorithm process as a progress on the plugin icon. To use this method you also have to declare this method to afford a successful compilation:

```
1 public event PluginProgressChangedEventHandler OnPluginProgressChanged
    ;
2 private void ProgressChanged(double value, double max)
3 {
4   EventsHelper.ProgressChanged(OnPluginProgressChanged, this, new
        PluginProgressEventArgs(value, max));
5 }
```

2.10 Perform a clean dispose

Be sure you have closed and cleaned all your streams after execution and when CrypTool decides to dispose the plugin instance. Though not required, we run the dispose code before execution as well:

```
1 public void Dispose()
2 {
3   foreach(CryptoolStream stream in listCryptoolStreamOut)
4   {
5     stream.Close();
6   }
7   listCryptoolStreamOut.Clear();
8 }
9
10 public void PostExecution()
11 {
12   Dispose();
13 }
14
15 public void PreExecution()
16 {
17   Dispose();
18 }
```

2.11 Finish implementation

When adding plugin instances to the CrypTool workspace, CrypTool checks whether the plugin runs without any exception. If any IPlugin method throws an exception, CrypTool will show an error and prohibit using the plugin. Therefore we have to remove the "NotImplementedException" from the methods "Initialize()", "Pause()" and "Stop()". In our example it's sufficient to provide empty implementations.

```
1 public void Initialize()
2 {
3 }
4
5 public void Pause()
6 {
7 }
8
9 public void Stop()
10 {
11 }
```

The methods "Presentation()" and "QuickWatchPresentation()" can be used if a plugin developer wants to provide an own visualization of the plugin algorithm which will be shown in CrypTool. Take a look at the PRESENT plugin to see how a custom visualization can be realized. For our Caesar example we don't want to implement a custom visualization, therefore we return "null":

```
1 public UserControl Presentation
2 {
3   get { return null; }
4 }
5
6 public UserControl QuickWatchPresentation
7 {
8   get { return null; }
9 }
```

Your plugin should compile without errors at this point.

2.12 Import the plugin to CrypTool and test it

After you have built the plugin, you need to move the newly created plugin DLL to a location, where CrypTool can find it. To do this, there are the following ways:

- Copy your plugin DLL file in the folder "CrypPlugins" which has to be in the same folder as the CrypTool executable, called "CrypWin.exe". If necessary, create the folder "CrypPlugins".
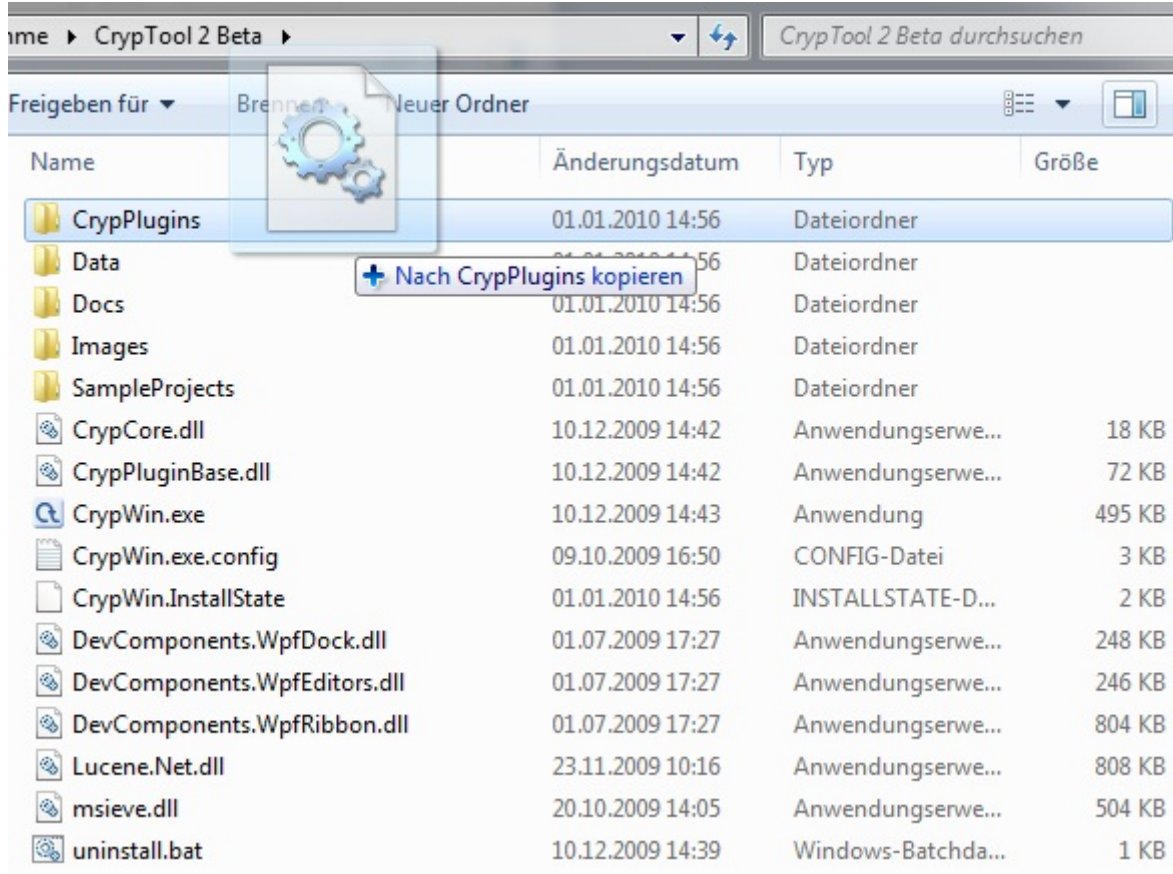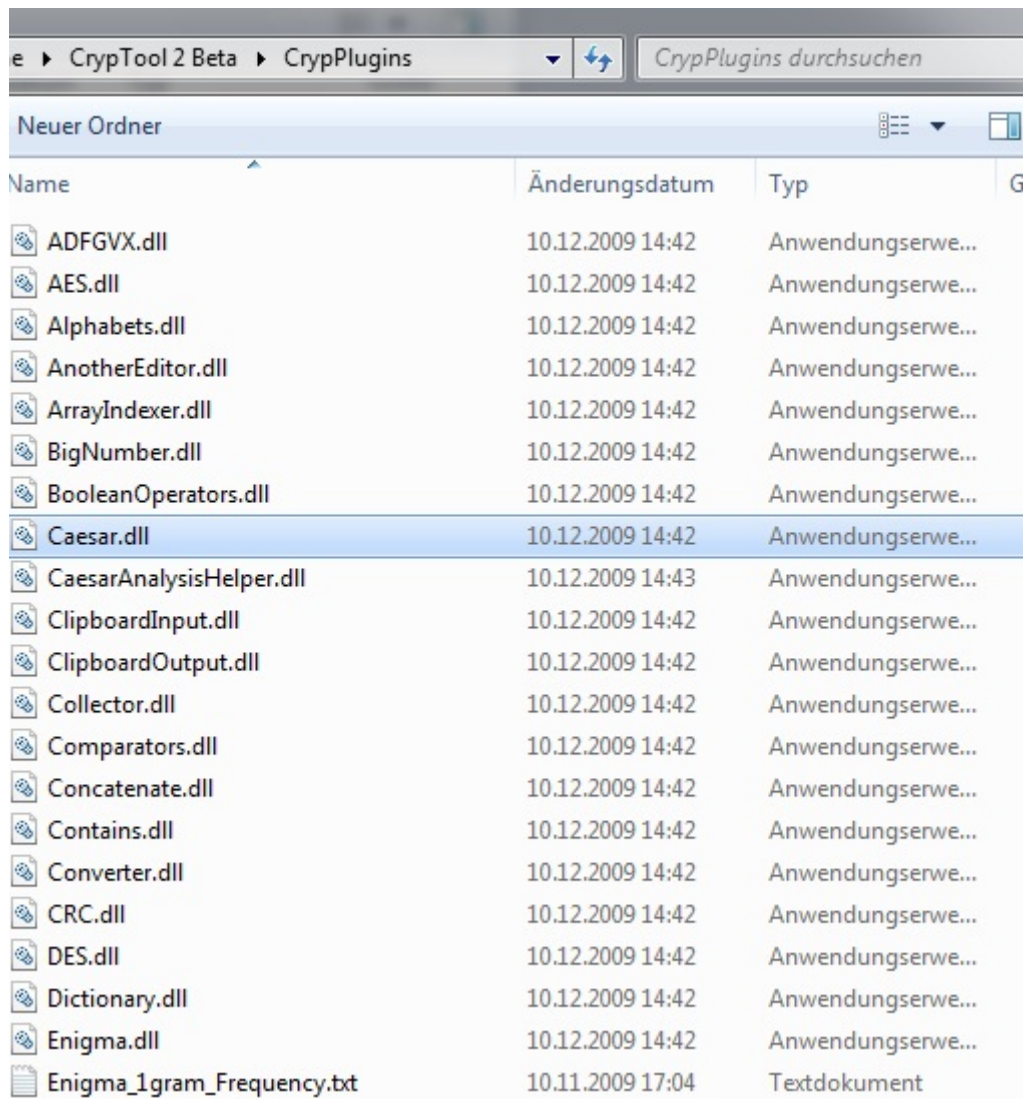

Figure 2.31: Copy plugin to global storage

This folder is called "Global storage" in the CrypTool architecture. Changes in this folder will take effect for all users on a multi user Windows. Finally restart CrypTool.

Figure 2.32: Plugins global storage

- Copy your plugin DLL file in the folder "CrypPlugins" which is located in your home path in the folder "ApplicationData" and restart CrypTool. This home folder path is called "Custom storage" in the CrypTool architecture. Changes in this folder will only take effect for current user. On a German Windows XP the home folder path could look like: "C:\Dokumente und Einstellungen\<User>\Anwendungsdaten\CrypPlugins" and in Vista/Windows7 the path will look like "C:\Users\<user>\Application Data\CrypPlugins".
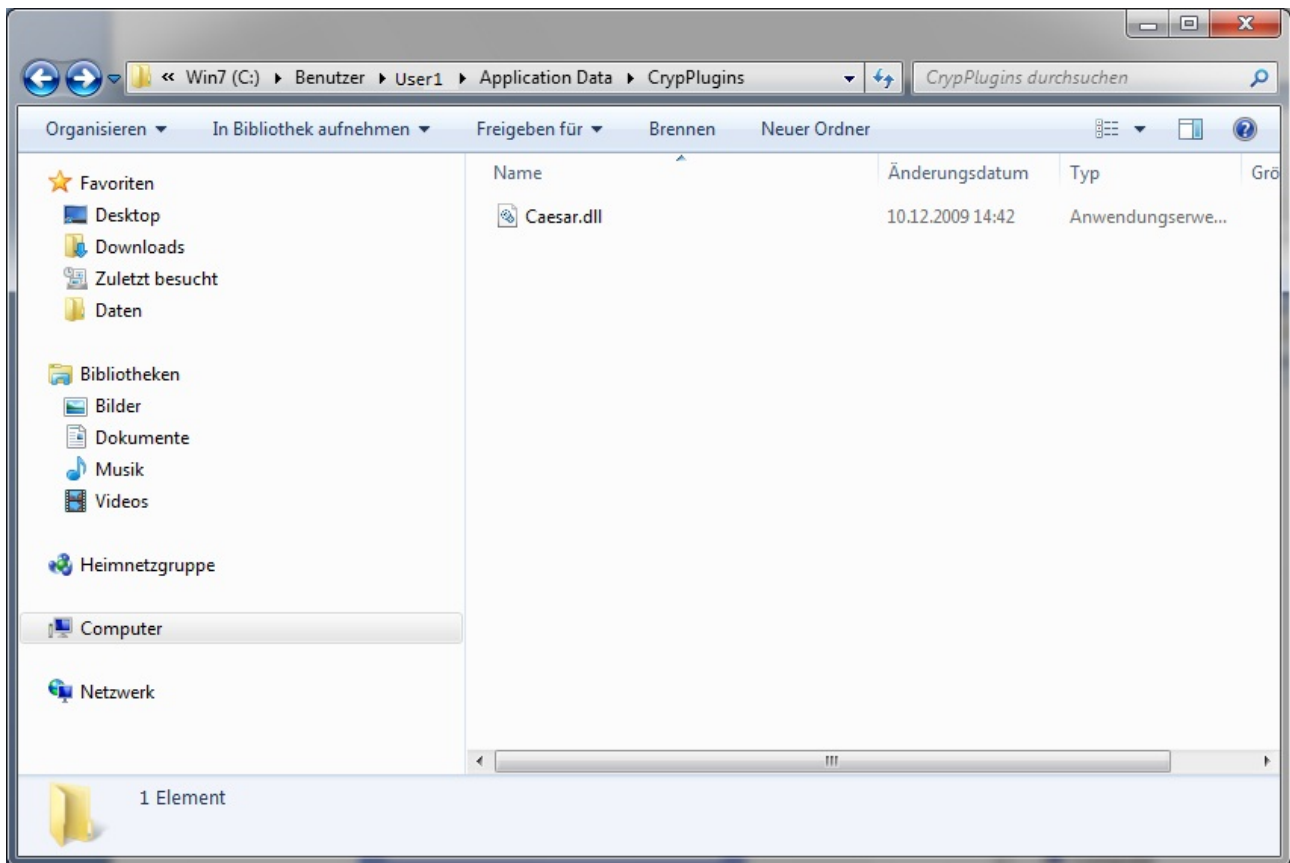
Figure 2.33: Plugins custom storage

- You can also import new plugins directly from the CrypTool interface. Just execute CrypWin.exe and select the "Download Plugins" button. An "Open File Dialog" will open and ask where the new plugin is located. After selecting the new plugin, CrypTool will automatically import the new plugin in the custom storage folder. With this option you will not have to restart CrypTool. All according menu entries will be updated automatically. Notice, that this plugin importing function only accepts **signed** plugins.

Note: This option is a temporary solution for importing new plugins. In the future this will be done online by a web service.

- Use post-build in your project properties to copy the DLL automatically after building it in Visual Studio with other plugins. Right-click on your plugin project and select "Properties":
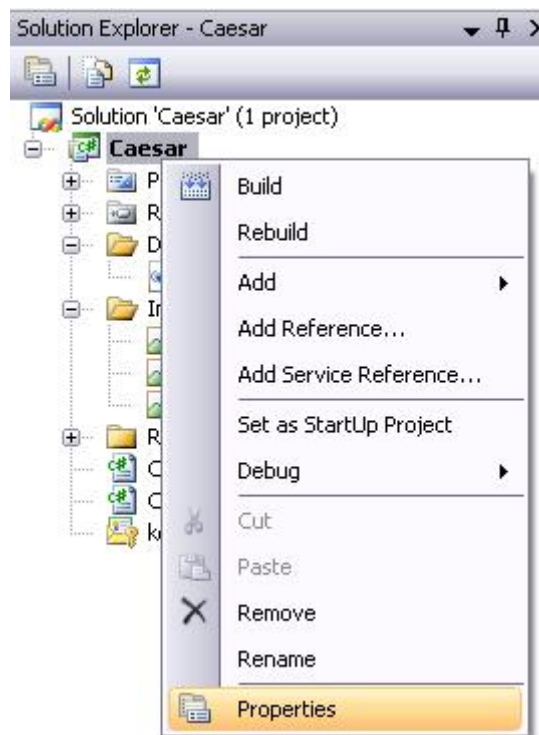

Figure 2.34: Solution Properties

Select "Build Events":

Figure 2.35: Build Events

Enter the following text snippet into "Post-build event command line":

cd "$(ProjectDir)"
cd ..\..\CrypWin$(OutDir)
if not exist "./CrypPlugins" mkdir "./CrypPlugins"
del /F /S /Q /s /q " Caesar *.*"
copy "$(TargetDir) Caesar *.*" "./CrypPlugins"

You need to adapt the yellow marked field to your actual project name.

2.13 Source code and source template

Here you can download the whole source code which was presented in this "Howto" as a Visual Studio **solution**:

username: anonymous
password: not required
https://www.cryptool.org/svn/CrypTool2/trunk/CrypPlugins/Caesar/

Here you can download the Visual Studio plugin **template** to begin with the development of a new CrypTool plugin:

http://cryptool2.vs.uni-due.de/downloads/template/encryptionplugin.zip

2.14 Provide a workflow file of your plugin

Every plugin developer should provide a workflow file which shows his algorithm working in CrypTool2. You will automatically create a workflow file by saving your project which was created on CrypTool2 work space. Here is an example how a workflow could look like:
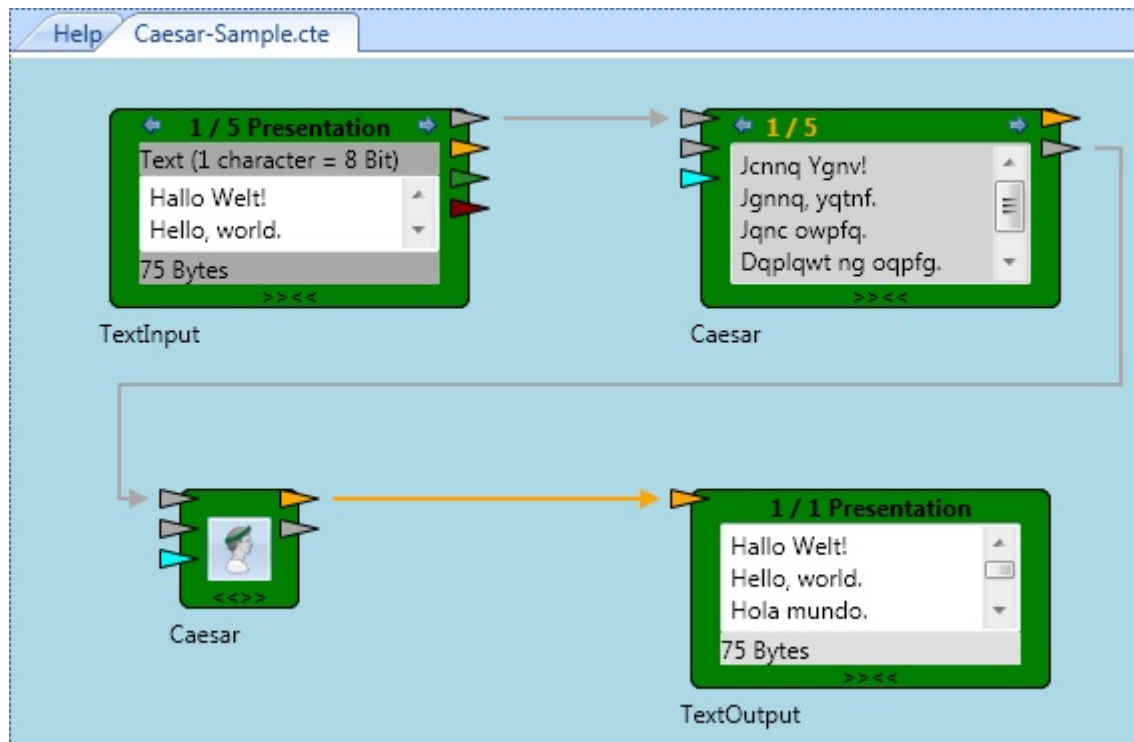

Figure 2.36: Plugin sample