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
March 9, 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.

To meet these ends, CrypTool 2 is built using the following:

- .NET (a modern software framework from Microsoft with solutions to common programming problems)
- C# (a modern object-oriented programming language, comparable to Java)
- WPF (a modern purely vector-based graphical subsystem for rendering user interfaces in Windows-based applications)
- Visual Studio 2008 (a development environment)
- 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 250,000 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.

1.2.1 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).

1.2.2 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.

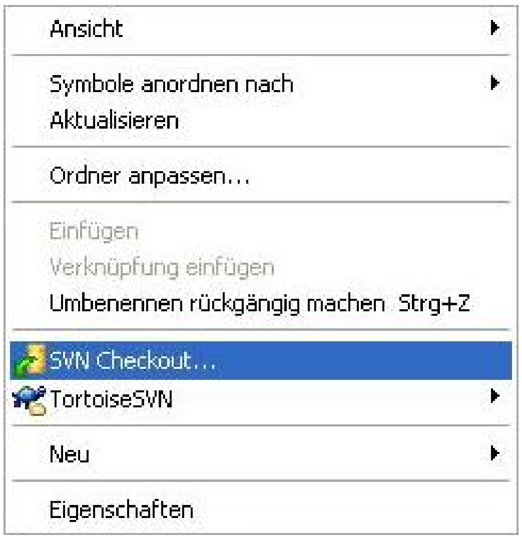


Figure 1.1: Selecting "SVN Checkout" from the context menu after installing TortoiseSVN.

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; use the address given above, as seen in Figure 1.2. 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.

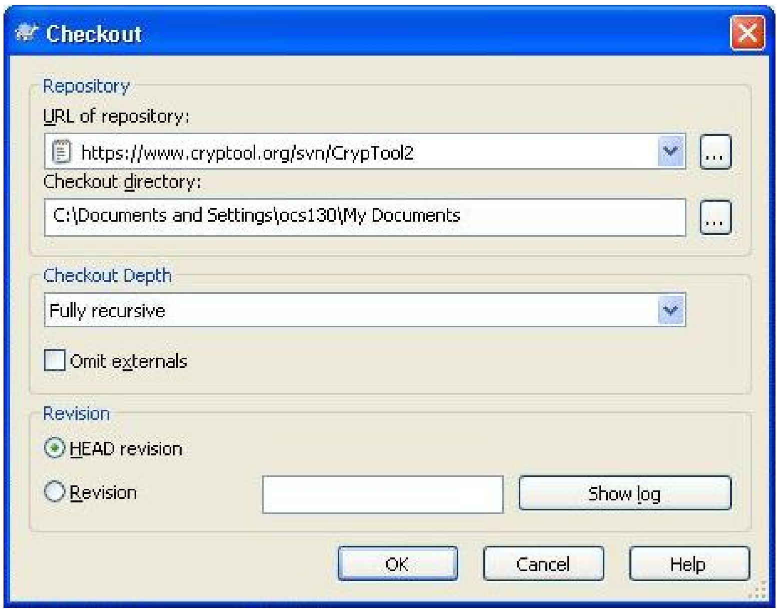


Figure 1.2: Checking out the CrypTool2 repository.

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.

1.2.3 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!

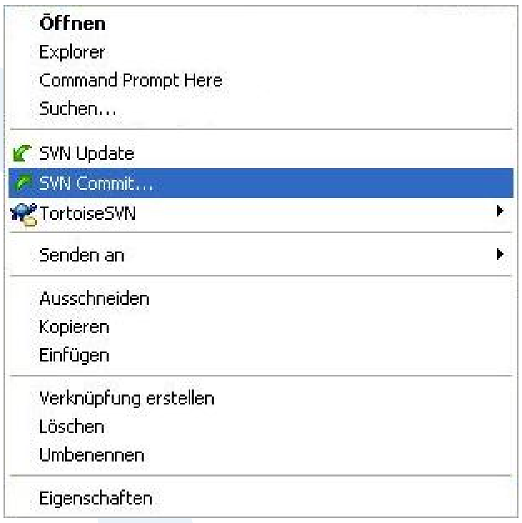


Figure 1.3: Selecting “SVN Commit” from the context menu.

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.

1.2.4 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.

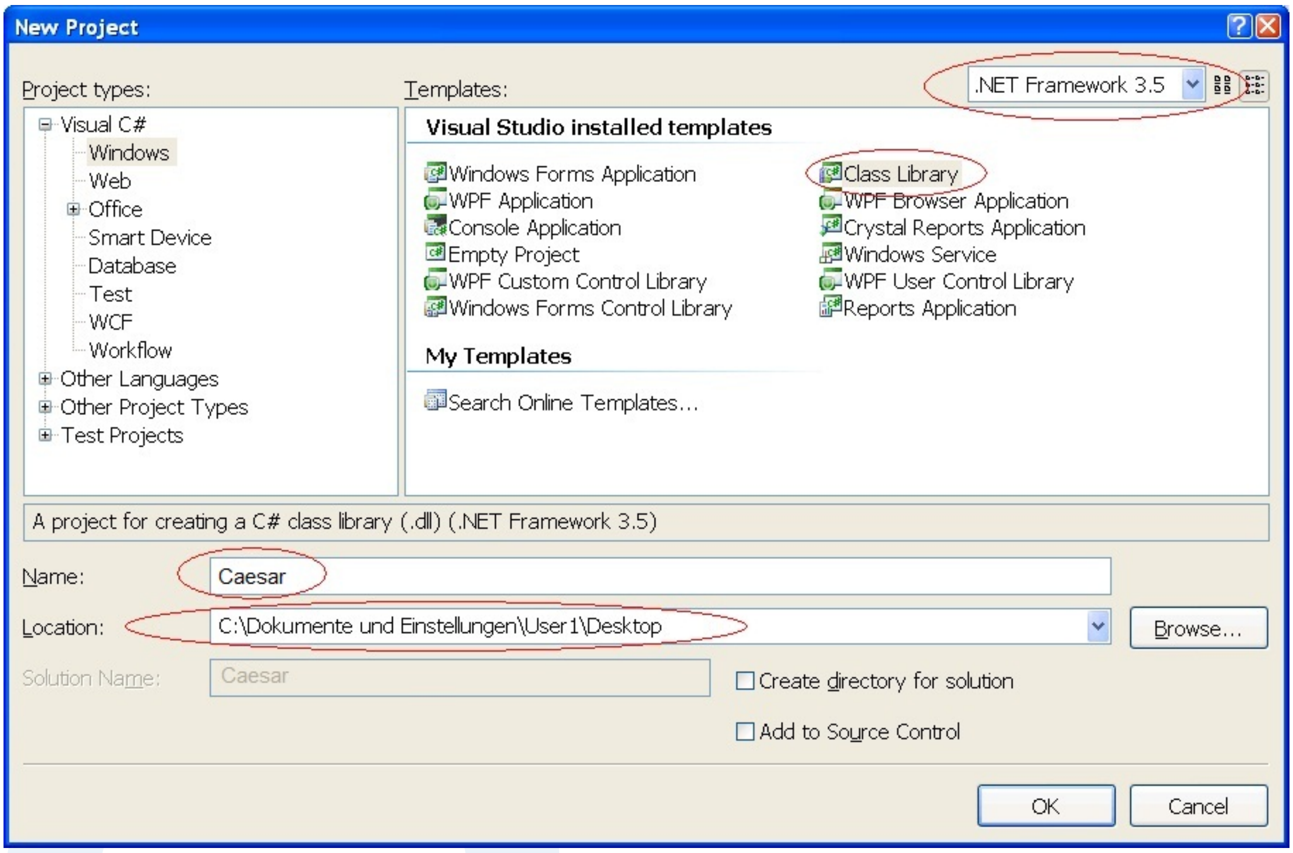
If you are a **core developer**, hence somebody who can also compile CryWin and AnotherEditor, you should use the ***CrypTool 2.0.sln*** solution from the *trunk/CoreDeveloper/* directory (which will *not* be visible to you if you are not a core developer). As a core developer, be aware that when you 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:



A project for creating a C# class library (.dll) (.NET Framework 3.5)

Name: Caesar

Location: C:\Dokumente und Einstellungen\User1\Desktop

Solution Name: Caesar

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.

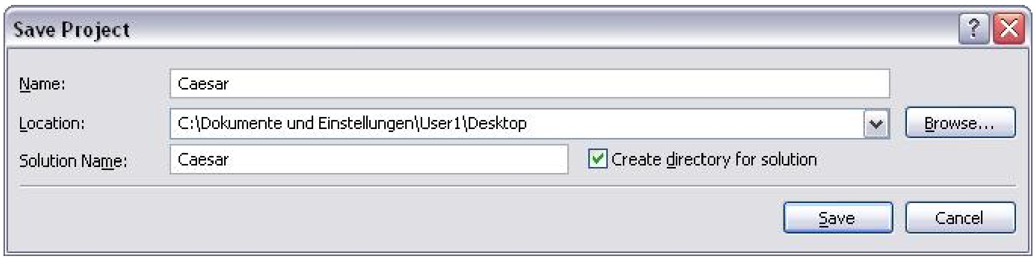
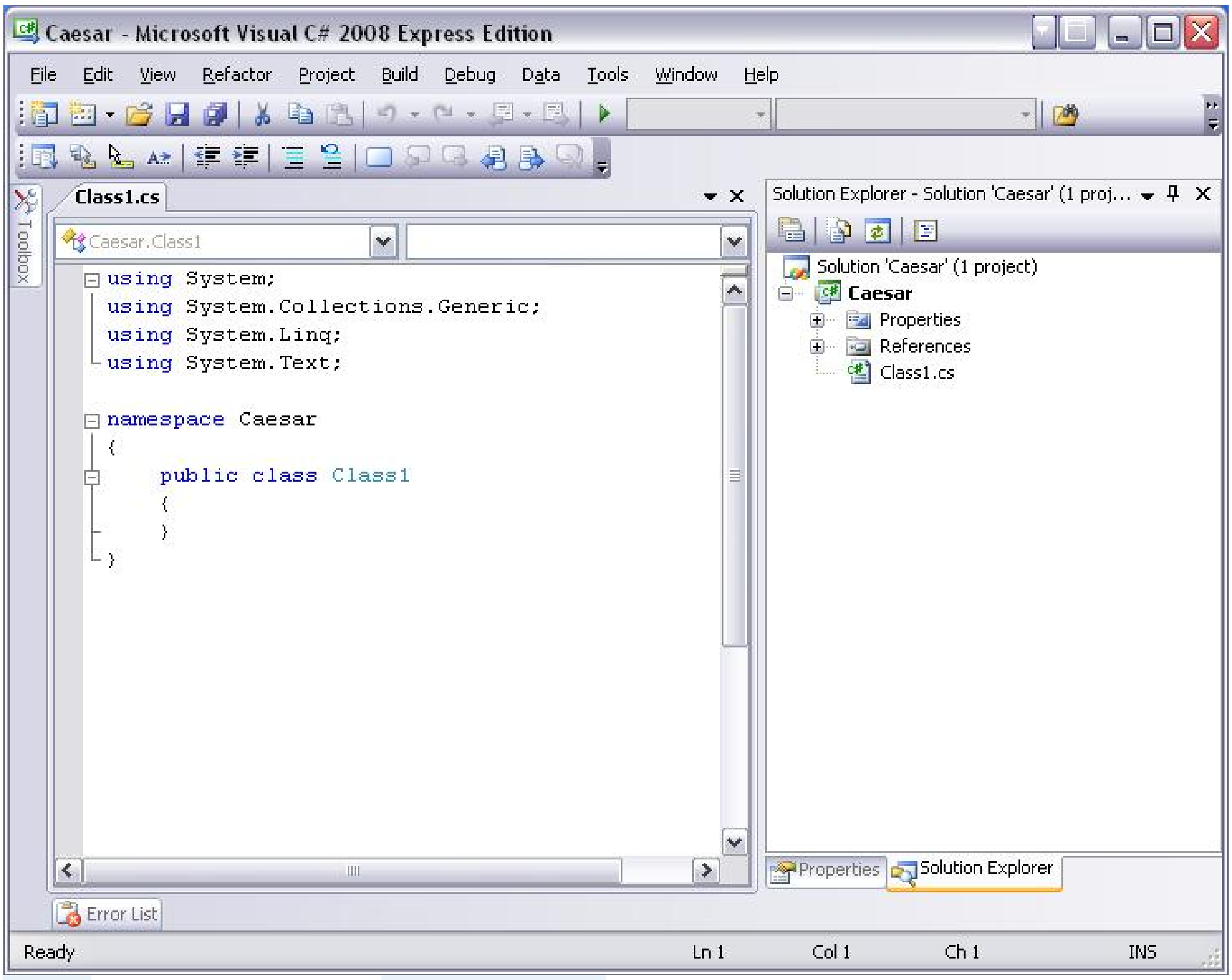

Save Project
Name: Caesar
Location: C:\Dokumente und Einstellungen\User1\Desktop
Browse...
Solution Name: Caesar
Create directory for solution
Save
Cancel

Figure 2.2: The Visual Studio C# Express Edition "Save Project" dialog window.

At this point, your Visual Studio/C# Express solution should look like this:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Caesar
{
    public class Class1
    {
    }
}
```

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.

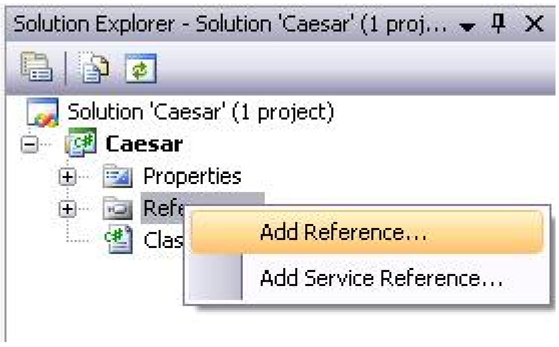
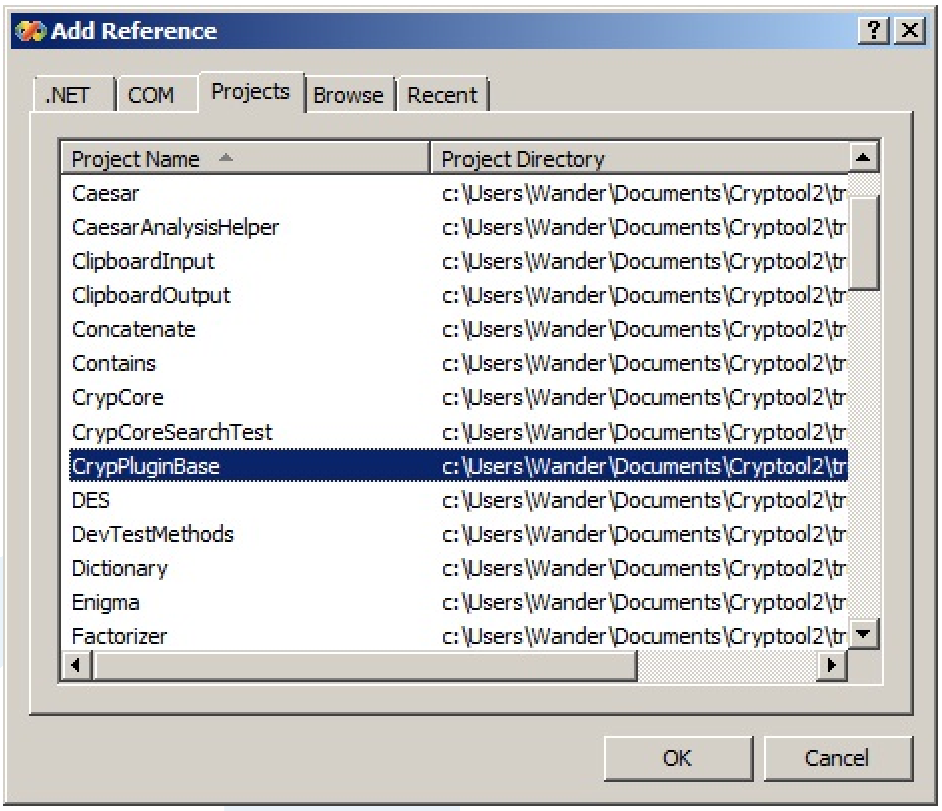

Solution Explorer - Solution 'Caesar' (1 proj...
Solution 'Caesar' (1 project)
Caesar
Properties
Refe
Clas
Add Reference...
Add Service Reference...

Figure 2.4: Adding a new reference.

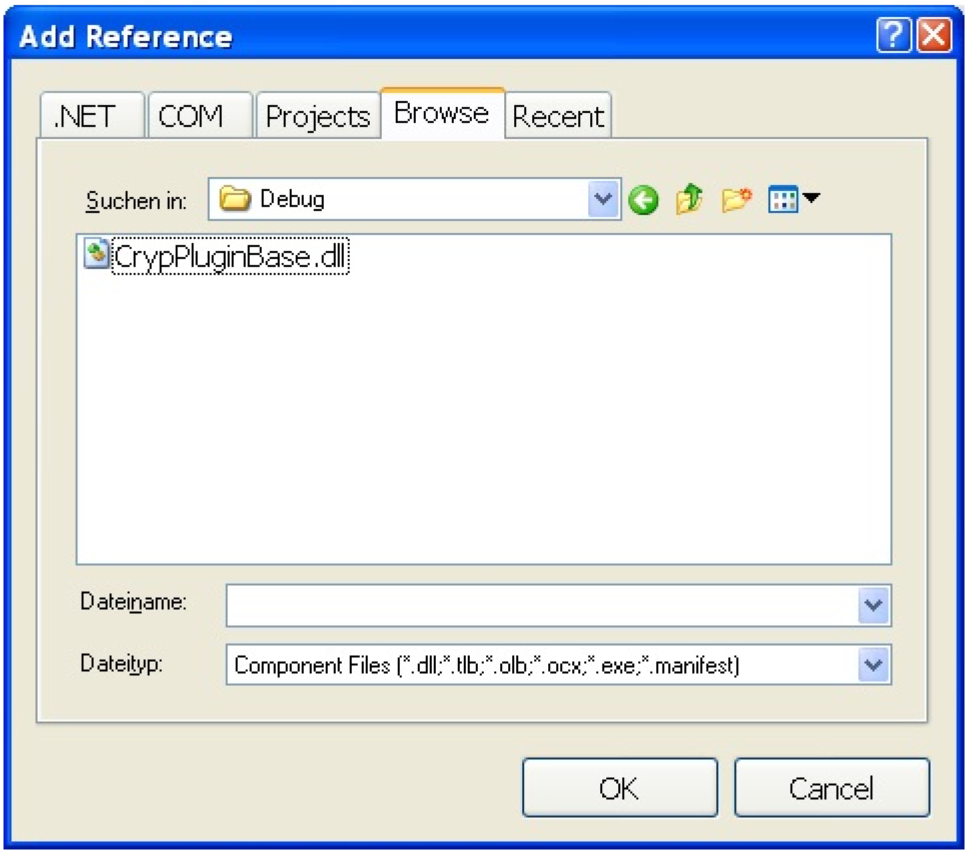
Right-click in the Solution Explorer on the "Reference" item and choose "Add Reference". A window like the following should appear:



Project Name	Project Directory
Caesar	c:\Users\Wander\Documents\Cryptool2\tr
CaesarAnalysisHelper	c:\Users\Wander\Documents\Cryptool2\tr
ClipboardInput	c:\Users\Wander\Documents\Cryptool2\tr
ClipboardOutput	c:\Users\Wander\Documents\Cryptool2\tr
Concatenate	c:\Users\Wander\Documents\Cryptool2\tr
Contains	c:\Users\Wander\Documents\Cryptool2\tr
CrypCore	c:\Users\Wander\Documents\Cryptool2\tr
CrypCoreSearchTest	c:\Users\Wander\Documents\Cryptool2\tr
CrypPluginBase	c:\Users\Wander\Documents\Cryptool2\tr
DES	c:\Users\Wander\Documents\Cryptool2\tr
DevTestMethods	c:\Users\Wander\Documents\Cryptool2\tr
Dictionary	c:\Users\Wander\Documents\Cryptool2\tr
Enigma	c:\Users\Wander\Documents\Cryptool2\tr
Factorizer	c:\Users\Wander\Documents\Cryptool2\tr

Figure 2.5: Adding a reference to the PluginBase source code.

Unless you have created your new project in the same CrypTool 2.0 solution, you probably will not be able to select the library directly as seen above in Figure 2.5; instead you can browse for the binary DLL as seen below in Figure 2.6. Click on the “Browse” tab and navigate to the folder in which you downloaded the CrypTool 2 project. Within that folder, go to *\CrypPluginBase\bin\Debug* and select the file “CryptPluginBase.dll”. The library reference can then be added by double clicking the file or pressing the “OK” button.



Suchen in: Debug

CrypPluginBase.dll

Dateiname:

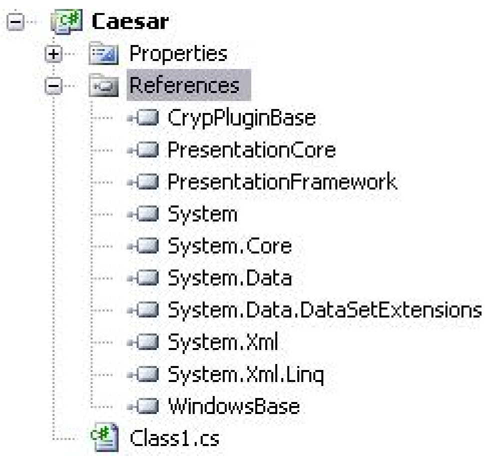
Dateityp: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

Figure 2.6: Browsing for a reference.

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

- PresentationCore
- PresentationFramework
- WindowsBase

Afterwards your reference tree view should look like this:



```
Caesar
  Properties
  References
    CrypPluginBase
    PresentationCore
    PresentationFramework
    System
    System.Core
    System.Data
    System.Data.DataSetExtensions
    System.Xml
    System.Xml.Linq
    WindowsBase
  Class1.cs
```

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:

```
[assembly: AssemblyVersion("2.0.*")]
//[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 encryption 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 will need 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".

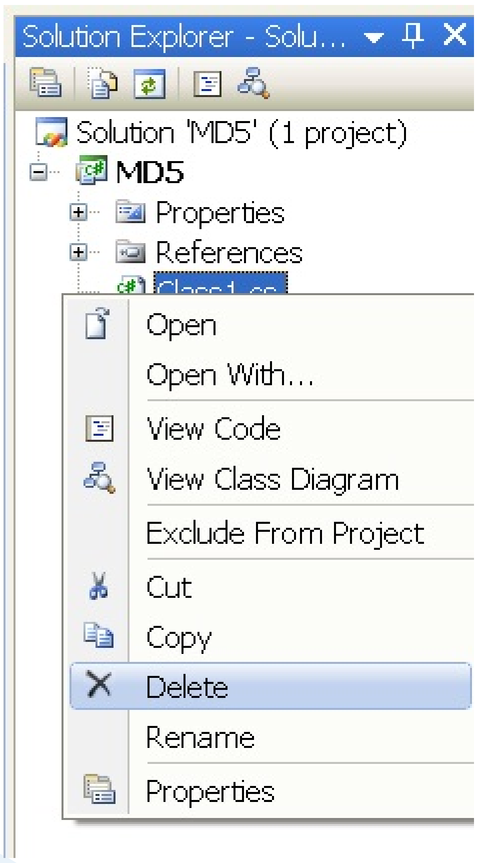


Figure 2.8: Deleting a class.

Then right-click on the project item (in our case, “Caesar”) and select “Add → Class. . . ”:

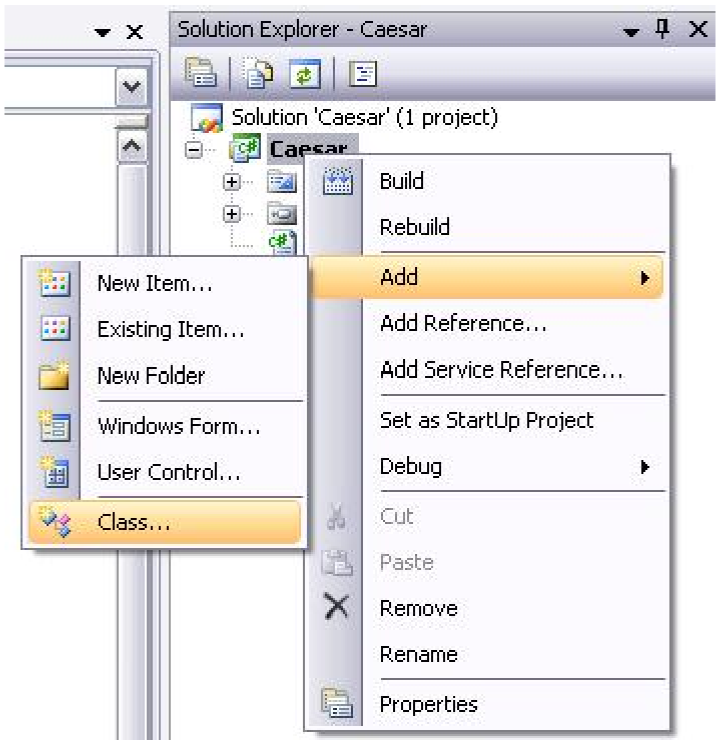
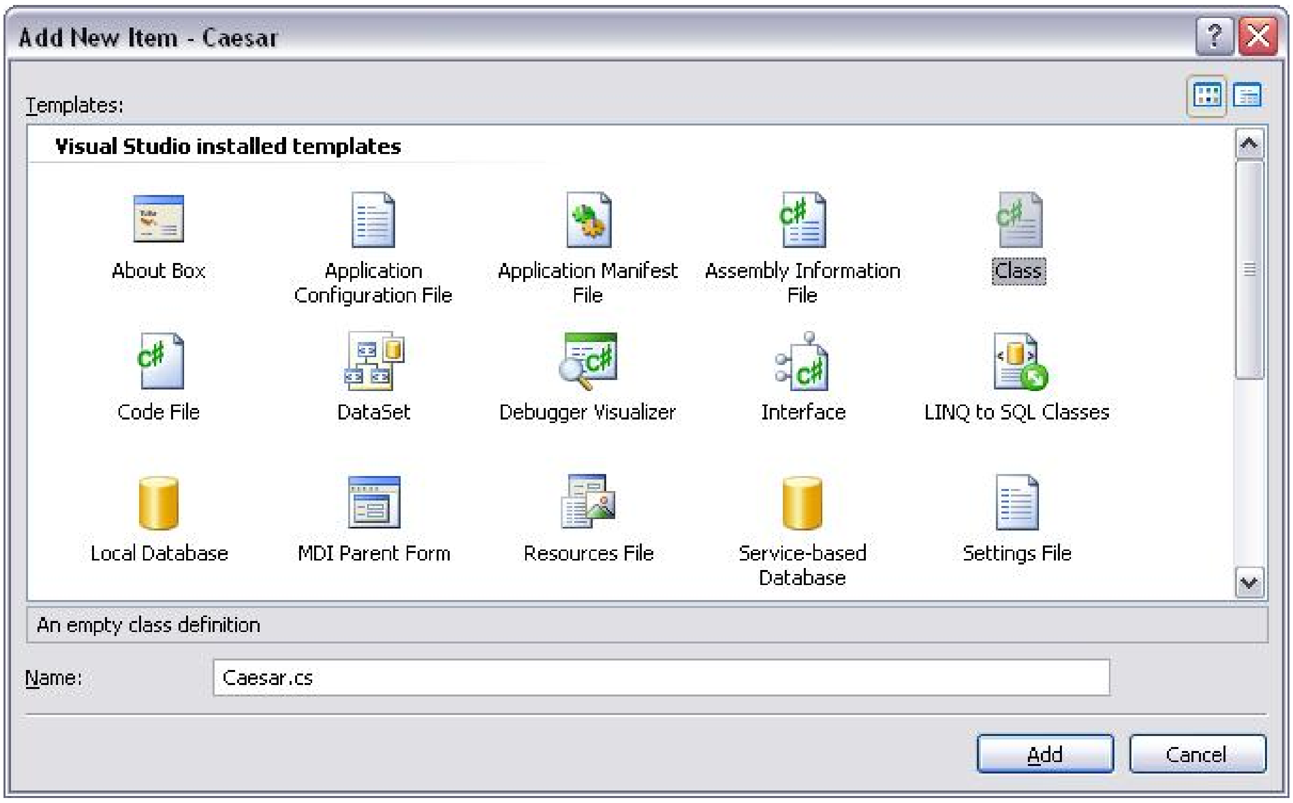


Figure 2.9: Adding a new class.

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.



An empty class definition

Name: Caesar.cs

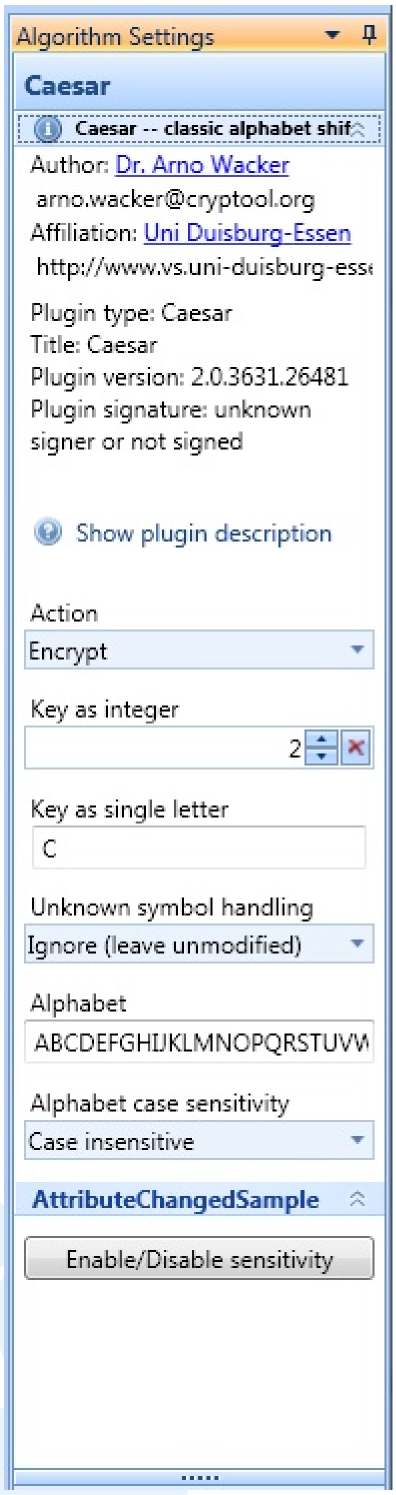
Figure 2.10: Naming the new class.

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:


Algorithm Settings

Caesar

Caesar -- classic alphabet shif

Author: Dr. Arno Wacker
arno.wacker@cryptool.org
Affiliation: Uni Duisburg-Essen
http://www.vs.uni-duisburg-esse

Plugin type: Caesar
Title: Caesar
Plugin version: 2.0.3631.26481
Plugin signature: unknown signer or not signed

Show plugin description

Action: Encrypt

Key as integer: 2

Key as single letter: C

Unknown symbol handling: Ignore (leave unmodified)

Alphabet: ABCDEFGHIJKLMNOPQRSTUVW

Alphabet case sensitivity: Case insensitive

AttributeChangedSample

Enable/Disable sensitivity

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 CrypTool 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 CrypTool 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:

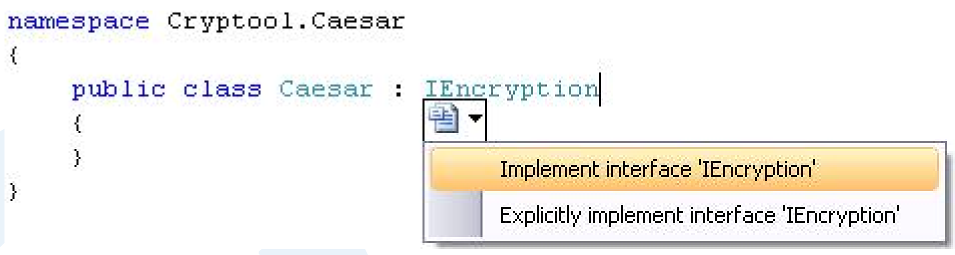
```
using System.Collections.Generic;
using System.Text;

//required CrypTool namespaces
using Cryptool.PluginBase;
using Cryptool.PluginBase.Cryptography;
using Cryptool.PluginBase.IO;
using Cryptool.PluginBase.Miscellaneous;

namespace Cryptool.Caesar
{
  public class Caesar
  {
  }
}
```

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

```
namespace Cryptool.Caesar
{
  public class Caesar : IEncryption
  {
  }
}
```

2.4.4 Adding interface functions to the Caesar class

You may notice an underscore underneath the "I" in "IEncryption". Move your mouse over it, or place the cursor on it and press "Shift+Alt+F10" and the following submenu should appear:



```
namespace Cryptool.Caesar
{
    public class Caesar : IEncryption
    {
    }
}
```

Implement interface 'IEncryption'
Explicitly implement interface 'IEncryption'

Figure 2.12: An inheritance submenu.

Select the item "Implement interface 'IEncryption'". Visual Studio will automatically generate all the interface members necessary for interaction with the CrypTool 2 core. (This step will save you a lot of typing!)

Your code should now look like this:

```
using System.Collections.Generic;
using System.Text;

using Cryptool.PluginBase;
using Cryptool.PluginBase.Cryptography;
using Cryptool.PluginBase.IO;
using Cryptool.PluginBase.Miscellaneous;

namespace Cryptool.Caesar
{
    public class Caesar : IEncryption
    {
        #region IPlugin Members

        public void Dispose()
        {
            throw new NotImplementedException();
        }

        public void Execute()
        {
            throw new NotImplementedException();
        }

        public void Initialize()
        {
            throw new NotImplementedException();
        }

        public event GuiLogNotificationEventHandler OnGuiLogNotificationOccured;

        public event PluginProgressChangedEventHandler OnPluginProgressChanged;

        public event StatusChangedEventHandler OnPluginStatusChanged;

        public void Pause()
        {
            throw new NotImplementedException();
        }

        public void PostExecution()
        {
            throw new NotImplementedException();
        }

        public void PreExecution()
        {
                throw new NotImplementedException();
            }

            public System.Windows.Controls.UserControl Presentation
            {
                get { throw new NotImplementedException(); }
            }

            public System.Windows.Controls.UserControl
               QuickWatchPresentation
            {
                get { throw new NotImplementedException(); }
            }

            public ISettings Settings
            {
                get { throw new NotImplementedException(); }
            }

            public void Stop()
            {
                throw new NotImplementedException();
            }

            #endregion

            #region INotifyPropertyChanged Members

            public event System.ComponentModel.PropertyChangedEventHandler
                PropertyChanged;

            #endregion
        }
}
```

2.4.5 Adding the namespace and interfaces to the CaesarSettings class

Let's now take a look at the second class in our example, "CaesarSettings", by double-clicking on the "CaesarSettings.cs" file in the Solution Explorer. First, we need to again include the "Cryptool.PluginBase" namespace to the class header. Then we must let the settings class inherit from "ISettings" in the same manner as was done with the Caesar class. Visual Studio will again automatically generate code from the CrypTool interface as seen below. (We can again remove the lines `using System;` and `using System.Linq;`, as we do not need those references.)

```
using System.Collections.Generic;
using System.Text;

using Cryptool.PluginBase;

```

```
namespace Cryptool.Caesar
{
    public class CaesarSettings : ISettings
    {
        #region ISettings Members

        public bool HasChanges
        {
            get
            {
                throw new NotImplementedException();
            }
            set
            {
                throw new NotImplementedException();
            }
        }

        #endregion

        #region INotifyPropertyChanged Members

        public event System.ComponentModel.PropertyChangedEventHandler
            PropertyChanged;

        #endregion
    }
}
```

2.4.6 Adding controls to the CaesarSettings class

The settings class is used to populate the TaskPane in the CrypTool 2 application so that the user can modify settings at will. To meet these ends we will need to implement some controls such as buttons and text boxes. If you will be implementing an algorithm that does not have any user-defined settings (e.g. a hash function), then this class can be left empty; you will, however, still have to modify the "HasChanges" property to avoid a "NotImplementedException". The following code demonstrates the modifications necessary to create the backend for the TaskPane for our Caesar algorithm. You can also look at the source code of other algorithms in the subversion repository for examples of how to create the TaskPane backend.

```
using System;
using System.ComponentModel;
using System.Windows;
using Cryptool.PluginBase;
using System.Windows.Controls;

namespace Cryptool.Caesar
{
    public class CaesarSettings : ISettings
    {
        #region Public Caesar specific interface

        /// <summary>
        /// We use this delegate to send log messages from
        /// the settings class to the Caesar plugin.
        /// </summary>
        public delegate void CaesarLogMessage(string msg,
           NotificationLevel loglevel);

        /// <summary>
        /// An enumeration for the different modes of handling
        /// unknown characters.
        /// </summary>
        public enum UnknownSymbolHandlingMode { Ignore = 0, Remove =
           1, Replace = 2 };

        /// <summary>
        /// Fires when a new status message was sent.
        /// </summary>
        public event CaesarLogMessage LogMessage;

        public delegate void CaesarReExecute();

        public event CaesarReExecute ReExecute;

        /// <summary>
        /// Retrieves or sets the current shift value (i.e. the key).
        /// </summary>
        [PropertySaveOrder(0)]
        public int ShiftKey
        {
            get { return shiftValue; }
            set
            {
                setKeyByValue(value);
            }
        }

        /// <summary>
        /// Retrieves the current setting of whether or not the
        /// alphabet should be treated as case-sensitive.
        /// </summary>
        [PropertySaveOrder(1)]
        public bool CaseSensitiveAlphabet
        {
            get
            {
                if (caseSensitiveAlphabet == 0)
                {   return false;   }
                else
                {   return true;    }
            }
            set {} // this setting is readonly, but we must include
                   // some form of set method to prevent problems.
        }


        /// <summary>
        /// Returns true if any settings have been changed.
        /// This value should be set externally to false e.g.
        /// when a project is saved.
        /// </summary>
        [PropertySaveOrder(3)]
        public bool HasChanges
        {
            get { return hasChanges; }
            set { hasChanges = value; }
        }

        #endregion

        #region Private variables
        private bool hasChanges;
        private int selectedAction = 0;
        private string upperAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        private string lowerAlphabet = "abcdefghijklmnopqrstuvwxyz";
        private string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        private char shiftChar = 'C';
        private int shiftValue = 2;
        private UnknownSymbolHandlingMode unknownSymbolHandling =
           UnknownSymbolHandlingMode.Ignore;
        private int caseSensitiveAlphabet = 0; // 0 = case-insensitve,
            1 = case-sensitive
        private bool sensitivityEnabled = true;
        #endregion

        #region Private methods

        private string removeEqualChars(string value)
        {
            int length = value.Length;

            for (int i = 0; i < length; i++)
            {
                for (int j = i + 1; j < length; j++)
                {
                    if ((value[i] == value[j]) || (!
                       CaseSensitiveAlphabet & (char.ToUpper(value[i])
                        == char.ToUpper(value[j]))))
                    {
                        LogMessage("Removing duplicate letter: \'" + value[j] + "\' from alphabet!", NotificationLevel.Warning);
                        value = value.Remove(j,1);
                        j--;
                        length--;
                    }
                }
            }

            return value;
        }

        /// <summary>
        /// Set the new shiftValue and the new shiftCharacter
        /// to offset % alphabet.Length.
        /// </summary>
        private void setKeyByValue(int offset)
        {
            HasChanges = true;

            // Make sure the shift value lies within the alphabet range.
            offset = offset % alphabet.Length;

            // Set the new shiftChar.
            shiftChar = alphabet[offset];

            // Set the new shiftValue.
            shiftValue = offset;

            // Announce this to the settings pane.
            OnPropertyChanged("ShiftValue");
            OnPropertyChanged("ShiftChar");

            // Print some info in the log.
            LogMessage("Accepted new shift value " + offset + "! (Adjusted shift character to \'" + shiftChar + "\')", NotificationLevel.Info);
        }

        private void setKeyByCharacter(string value)
        {
            try
            {
                int offset;
                if (this.CaseSensitiveAlphabet)
                {
                    offset = alphabet.IndexOf(value[0]);
                }
                else
                {
                    offset = alphabet.ToUpper().IndexOf(char.ToUpper(value[0]));
                }

                if (offset >= 0)
                {
                    HasChanges = true;
                    shiftValue = offset;
                    shiftChar = alphabet[shiftValue];
                    LogMessage("Accepted new shift character \'" + shiftChar + "\'! (Adjusted shift value to " + shiftValue + ")", NotificationLevel.Info);
                    OnPropertyChanged("ShiftValue");
                    OnPropertyChanged("ShiftChar");
                }
                else
                {
                    LogMessage("Bad input \"" + value + "\"! (Character not in alphabet!) Reverting to " + shiftChar.ToString() + "!", NotificationLevel.Error);
                }
            }
            catch (Exception e)
            {
                LogMessage("Bad input \"" + value + "\"! (" + e.Message + ") Reverting to " + shiftChar.ToString() + "!", NotificationLevel.Error);
            }
        }

        #endregion

        #region Algorithm settings properties (visible in the Settings pane)

        [PropertySaveOrder(4)]
        [ContextMenu("Action", "Select the algorithm action", 1, DisplayLevel.Beginner, ContextMenuControlType.ComboBox, new int[] { 1, 2 }, "Encrypt", "Decrypt")]
        [TaskPane("Action", "setAlgorithmActionDescription", null, 1, true, DisplayLevel.Beginner, ControlType.ComboBox, new string[] { "Encrypt", "Decrypt" })]
        public int Action
        {
            get
            {
                return this.selectedAction;
            }
            set
            {
                if (value != selectedAction) HasChanges = true;
                this.selectedAction = value;
                OnPropertyChanged("Action");

                if (ReExecute != null) ReExecute();
            }
        }

        [PropertySaveOrder(5)]
        [TaskPane("Key as integer", "Enter the number of letters to
           shift. For example, 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)]
        public int ShiftValue
        {
            get { return shiftValue; }
            set
            {
                setKeyByValue(value);
                if (ReExecute != null) ReExecute();
            }
        }

        [PropertySaveOrder(6)]
        [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}")]
        public string ShiftChar
        {
            get { return this.shiftChar.ToString(); }
            set
            {
                setKeyByCharacter(value);
                if (ReExecute != null) ReExecute();
            }
        }

        [PropertySaveOrder(7)]
        [ContextMenu("Unknown symbol handling", "What should be done
           with characters encountered in the input which are not in
```

```
            the alphabet?", 4, DisplayLevel.Expert,
            ContextMenuControlType.ComboBox, null, new string[] { "
            Ignore (leave unmodified)", "Remove", "Replace with \'?\'"
            })]
        [TaskPane("Unknown symbol handling", "What should be done with
             characters encountered in the input which are not in the
            alphabet?", null, 4, true, DisplayLevel.Expert, ControlType
            .ComboBox, new string[] { "Ignore (leave unmodified)", "
            Remove", "Replace with \'?\'" })]
        public int UnknownSymbolHandling
        {
            get { return (int)this.unknownSymbolHandling; }
            set
            {
                if ((UnknownSymbolHandlingMode)value !=
                   unknownSymbolHandling) HasChanges = true;
                this.unknownSymbolHandling = (
                   UnknownSymbolHandlingMode)value;
                OnPropertyChanged("UnknownSymbolHandling");

                if (ReExecute != null) ReExecute();
            }
        }

        [SettingsFormat(0, "Normal", "Normal", "Black", "White",
            Orientation.Vertical)]
        [PropertySaveOrder(9)]
        [TaskPane("Alphabet", "This is the alphabet currently in use."
            , null, 6, true, DisplayLevel.Expert, ControlType.TextBox,
            "")]
        public string AlphabetSymbols
        {
          get { return this.alphabet; }
          set
          {
            string a = removeEqualChars(value);
            if (a.Length == 0) // cannot accept empty alphabets
            {
              LogMessage("Ignoring empty alphabet from user! Using
                 previous alphabet instead: \"" + alphabet + "\" (" +
                 alphabet.Length.ToString() + " Symbols)",
                 NotificationLevel.Info);
            }
            else if (!alphabet.Equals(a))
            {
              HasChanges = true;
              this.alphabet = a;
              setKeyByValue(shiftValue); // reevaluate if the
                 shiftvalue is still within the range
              LogMessage("Accepted new alphabet from user: \"" +
                alphabet + "\" (" + alphabet.Length.ToString() + " 
                Symbols)", NotificationLevel.Info);
            OnPropertyChanged("AlphabetSymbols");

            if (ReExecute != null) ReExecute();
          }
        }
      }

      /// <summary>
      /// Visible setting how to deal with alphabet case.
      /// 0 = case-insentive, 1 = case-sensitive
      /// </summary>
      //[SettingsFormat(1, "Normal")]
      [PropertySaveOrder(8)]
      [ContextMenu("Alphabet case sensitivity", "Should upper and
         lower case be treated as the same (so that 'a' = 'A')?", 7,
          DisplayLevel.Expert, ContextMenuControlType.ComboBox, null
         , new string[] { "Case insensitive", "Case sensitive" })]
      [TaskPane("Alphabet case sensitivity", "Should upper and lower
          case be treated as the same (so that 'a' = 'A')?", null,
         7, true, DisplayLevel.Expert, ControlType.ComboBox, new
         string[] { "Case insensitive", "Case sensitive" })]
      public int AlphabetCase
      {
          get { return this.caseSensitiveAlphabet; }
          set
          {
              if (value != caseSensitiveAlphabet) HasChanges = true;
              this.caseSensitiveAlphabet = value;
              if (value == 0)
              {
                  if (alphabet == (upperAlphabet + lowerAlphabet))
                  {
                      alphabet = upperAlphabet;
                      LogMessage("Changing alphabet to: \"" +
                         alphabet + "\" (" + alphabet.Length.
                         ToString() + " Symbols)", NotificationLevel
                         .Info);
                      OnPropertyChanged("AlphabetSymbols");
                      // reset the key (shiftvalue/shiftChar)
                      // to be in the range of the new alphabet.
                      setKeyByValue(shiftValue);
                  }
              }
              else
              {
                  if (alphabet == upperAlphabet)
                  {
                      alphabet = upperAlphabet + lowerAlphabet;
```

```
                    LogMessage("Changing alphabet to: \"" +
                       alphabet + "\" (" + alphabet.Length.
                       ToString() + " Symbols)", NotificationLevel
                       .Info);
                    OnPropertyChanged("AlphabetSymbols");
                }
            }

            // Remove equal characters from the current alphabet.
            string a = alphabet;
            alphabet = removeEqualChars(alphabet);
            if (a != alphabet)
            {
                OnPropertyChanged("AlphabetSymbols");
                LogMessage("Changing alphabet to: \"" + alphabet +
                    "\" (" + alphabet.Length.ToString() + "
                   Symbols)", NotificationLevel.Info);
            }
            OnPropertyChanged("AlphabetCase");
            if (ReExecute != null) ReExecute();
        }
    }

    #endregion

    #region INotifyPropertyChanged Members

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string name)
    {
      if (PropertyChanged != null)
      {
        PropertyChanged(this, new PropertyChangedEventArgs(name));
      }
    }

    #endregion

    #region TaskPaneAttributeChanged (Sample)
    /// <summary>
    /// This event is here merely as a sample.
    /// </summary>
    public event TaskPaneAttributeChangedHandler
       TaskPaneAttributeChanged;

    [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)]
```

```
        public void EnableDisableSesitivity()
        {
          if (TaskPaneAttributeChanged!= null)
          {
            sensitivityEnabled = !sensitivityEnabled;
            if (sensitivityEnabled)
            {
              TaskPaneAttributeChanged(this, new TaskPaneAttributeChangedEventArgs(new TaskPaneAttribteContainer("AlphabetCase", Visibility.Visible)));
            }
            else
            {
              TaskPaneAttributeChanged(this, new TaskPaneAttributeChangedEventArgs(new TaskPaneAttribteContainer("AlphabetCase", Visibility.Collapsed)));
            }
          }
        }
        #endregion TaskPaneAttributeChanged (Sample)
    }
}
```

2.5 Adding an icon to the Caesar class

Before we go back to the code of the Caesar class, we have to add an icon to our project, which will be shown in the CrypTool **ribbon bar** and **navigation pane**. As there is currently no default, it is mandatory to add an icon. (It is planned to include a default icon in future versions.)

For testing purposes you can just create a simple black and white PNG image with MS Paint or Paint.NET. The proper image size is 40x40 pixels, but since the image will be rescaled if necessary, any size is technically acceptable.

Once you have saved your icon, you should add it directly to the project or to a subdirectory. In the project solution, we created a new folder named "Images". This can be done by right-clicking on the project item ("Caesar" in our example) and selecting "Add → New Folder". The icon can be added to this folder (or to the project directly, or to any other subdirectory) by right-clicking on the folder and selecting "Add → Existing Item".

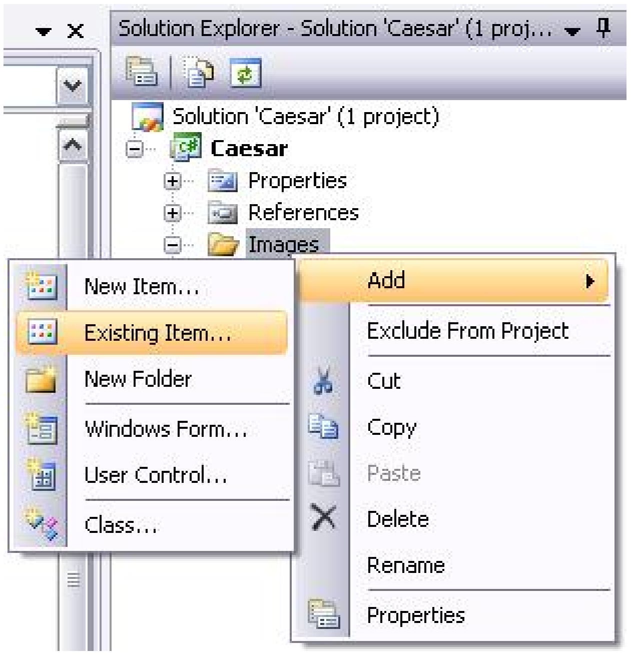


Figure 2.13: Adding an existing item.

A new window will then appear. Select “Image Files” as the file type and select your newly-created icon for your plugin.

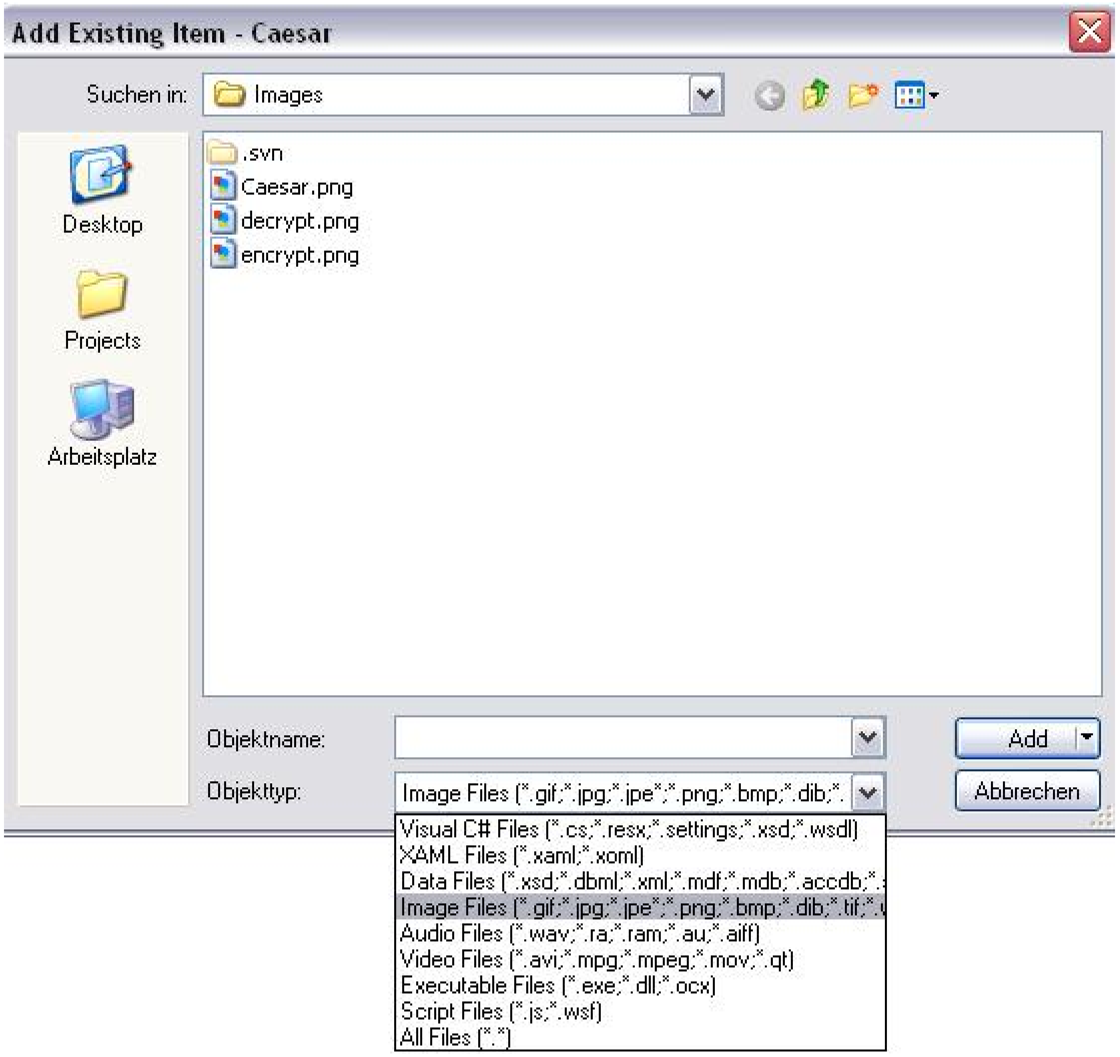

Add Existing Item - Caesar
Suchen in:
Images
Desktop
Projects
Arbeitsplatz
.svn
Caesar.png
decrypt.png
encrypt.png
Objektname:
Objekttyp:
Image Files (*.gif;*.jpg;*.jpe*;*.png;*.bmp;*.dib;*.
Add
Abbrechen
Visual C# Files (*.cs;*.resx;*.settings;*.xsd;*.wsdl)
XAML Files (*.xaml;*.xoml)
Data Files (*.xsd;*.dbml;*.xml;*.mdf;*.mdb;*.accdb;*.
Image Files (*.gif;*.jpg;*.jpe*;*.png;*.bmp;*.dib;*.tif;*.
Audio Files (*.wav;*.ra;*.ram;*.au;*.aiff)
Video Files (*.avi;*.mpg;*.mpeg;*.mov;*.qt)
Executable Files (*.exe;*.dll;*.ocx)
Script Files (*.js;*.wsf)
All Files (*.*)

Figure 2.14: Selecting the image file.

Finally, we must set the icon as a "Resource" to avoid including the icon as a separate file. Right-click on the icon and select "Properties" as seen below.

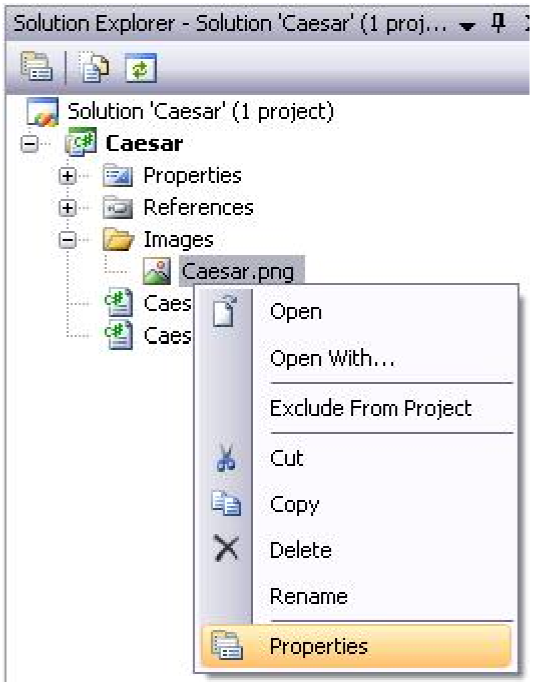


Figure 2.15: Selecting the image properties.

In the "Properties" panel, set the "Build Action" to "Resource".

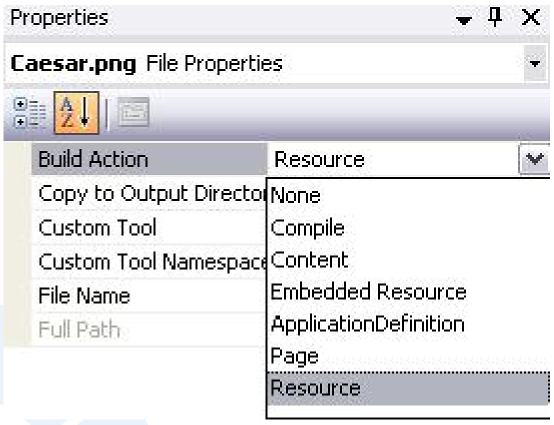


Figure 2.16: Selecting the icon's build action.

2.6 Defining the attributes of the Caesar class

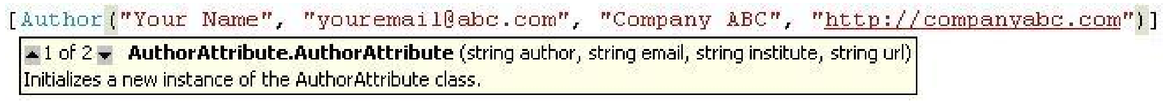
Now let's go back to the code of the Caesar class (the "Caesar.cs" file in our example). The first thing we will do is define the attributes of our class. These attributes are used to provide additional information for the CrypTool 2.0 environment. If they are not properly defined, your plugin won't show up in the application display, even if everything else is implemented correctly.

Attributes are used for **declarative** programming and provide metadata that can be added to the existing .NET metadata. CrypTool provides a set of custom attributes that are used to mark the different parts of your plugin.

These attributes can be defined anywhere within the "Cryptool.Caesar" namespace, but customarily they are defined right before the class declaration.

2.6.1 The *[Author]* attribute

The *[Author]* attribute is optional, and thus we are not required to define it. The attribute can be used to provide additional information about the plugin developer. This information will appear in the TaskPane, as for example in Figure 2.11. We will define the attribute to demonstrate how it should look in case you want to use it in your plugin.



```
[Author("Your Name", "youremail@abc.com", "Company ABC", "http://companyabc.com")]
```

```
▲1 of 2▼ AuthorAttribute.AuthorAttribute (string author, string email, string institute, string url)
Initializes a new instance of the AuthorAttribute class.
```

Figure 2.17: The defintion for the *[Author]* attribute.

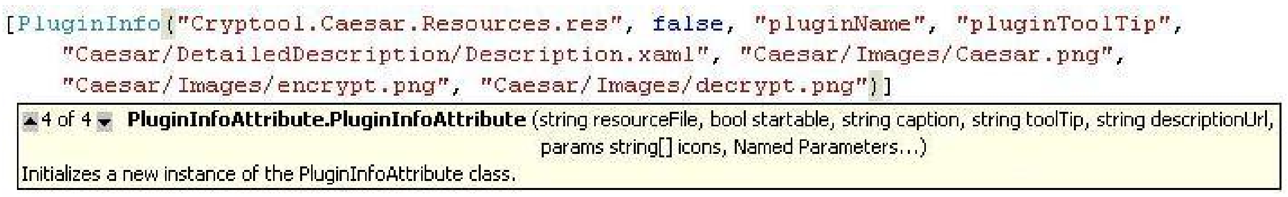
As can be seen above, the author attribute takes four elements of type string. These elements are:

- Author — the name of the plugin developer(s).
- Email — the email address of the plugin developer(s), should they wish to be available for contact.
- Institute — the organization, company or university with which the developer(s) are affiliated.
- URL — the website of the developer(s) or their institution.

All of these elements are optional; the developer(s) can choose what information will be published. Unused elements should be set to null or an empty string.

2.6.2 The *[PluginInfo]* attribute

The second attribute, *[PluginInfo]*, provides necessary information about the plugin, and is therefore mandatory. This information appears in the caption and tool tip window. The attribute is defined as follows:



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

```
4 of 4  PluginInfoAttribute.PluginInfoAttribute (string resourceFile, bool startable, string caption, string toolTip, string descriptionUrl,
                                                  params string[] icons, Named Parameters...)
Initializes a new instance of the PluginInfoAttribute class.
```

Figure 2.18: The defintion for the *[PluginInfo]* attribute.

This attribute has the following parameters:

- Resource File — defines where to find the associated resource file, if one is to be implemented. These are used, for example, to provide multilingual support for the plugin. This element is optional.
- Startable — a flag that should be set to true only if the plugin is an input generator plugin (i.e. if your plugin only has outputs and no inputs). In all other cases this should be set to false. This flag is important — setting it incorrectly will result in unpredictable results. This element is mandatory.
- Caption — the name of the plugin or, if the caption is specified in a resource file, the name of the appropriate field in the resource file. This element is mandatory.
- ToolTip — a description of the plugin or, if the tool tip is specified in a resource file, the name of the appropriate field in the resource file. This element is optional.
- DescriptionURL — defines where to find the description file (e.g. XAML file). This element is optional.
- Icons — an array of strings to define all the paths for the icons to be used in the plugin (i.e. the plugin icon described in section 2.5). This element is mandatory.

Unused elements should be set to null or an empty string.

(There are a few limitations and bugs that still exist in the *[PluginInfo]* attribute that will be resolved in a future version. Firstly, it is possible to use the plugin without setting a caption, although this is not recommended. In the future the plugin will fail to load without a caption. Secondly, a zero-length toolTip string currently causes the toolTip to appear as an empty box in the application. Lastly, the toolTip and description do not currently support internationalization and localization.)

In our example, the "resourceFile" parameter should be set to "Cryptool.Caesar.Resource.res". This file will be used to store the label and caption text to support multilingualism.

The second parameter, "startable" should be set to "false", because our encryption algorithm is not an input generator plugin.

The next two parameters are necessary to define the plugin's name and description. Since we are using a resource file, we should place here the names of the resource fields that contain the description and caption. (We call also just write simple text strings instead of using outsourced references.)

The next element defines the location path of the description file. The parameter is composed in the format <*assembly name*>/<*file name*> or, if you want to store your description files in a separate folder (as in our case), <*assembly name*>/<*path*>/<*file name*>. The description file must be an

XAML file. In our case, we shall create a folder named “DetailedDescription” in which to store our XAML file with any necessary images. Our folder structure looks as follows:

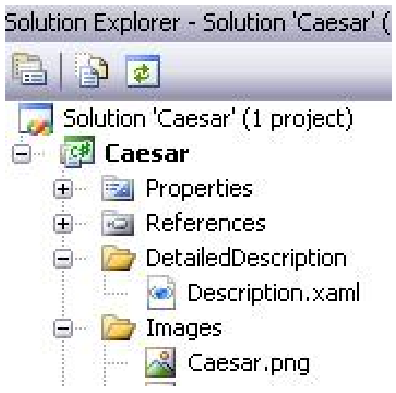
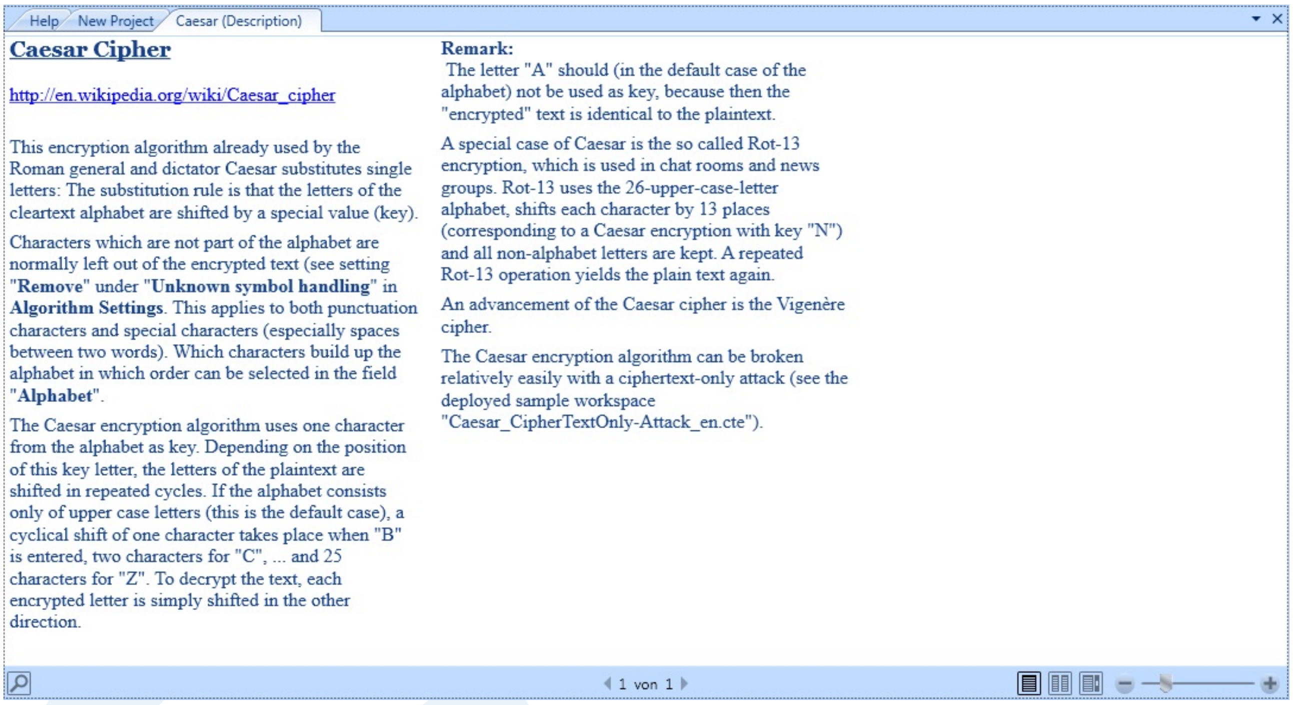


Figure 2.19: The folder structure as seen in the Solution Explorer.

Once a detailed description has been written in the XAML file, it can be accessed in the CrypTool 2 application by right-clicking on the plugin icon in the workspace and selecting “Show description”.



Caesar Cipher

http://en.wikipedia.org/wiki/Caesar_cipher

This encryption algorithm already used by the Roman general and dictator Caesar substitutes single letters: The substitution rule is that the letters of the cleartext alphabet are shifted by a special value (key).

Characters which are not part of the alphabet are normally left out of the encrypted text (see setting "**Remove**" under "**Unknown symbol handling**" in **Algorithm Settings**. This applies to both punctuation characters and special characters (especially spaces between two words). Which characters build up the alphabet in which order can be selected in the field "**Alphabet**".

The Caesar encryption algorithm uses one character from the alphabet as key. Depending on the position of this key letter, the letters of the plaintext are shifted in repeated cycles. If the alphabet consists only of upper case letters (this is the default case), a cyclical shift of one character takes place when "B" is entered, two characters for "C", ... and 25 characters for "Z". To decrypt the text, each encrypted letter is simply shifted in the other direction.

Remark:
The letter "A" should (in the default case of the alphabet) not be used as key, because then the "encrypted" text is identical to the plaintext.

A special case of Caesar is the so called Rot-13 encryption, which is used in chat rooms and news groups. Rot-13 uses the 26-upper-case-letter alphabet, shifts each character by 13 places (corresponding to a Caesar encryption with key "N") and all non-alphabet letters are kept. A repeated Rot-13 operation yields the plain text again.

An advancement of the Caesar cipher is the Vigenère cipher.

The Caesar encryption algorithm can be broken relatively easily with a ciphertext-only attack (see the deployed sample workspace "Caesar_CipherTextOnly-Attack_en.cte").

Figure 2.20: A detailed description provided through an XAML file.

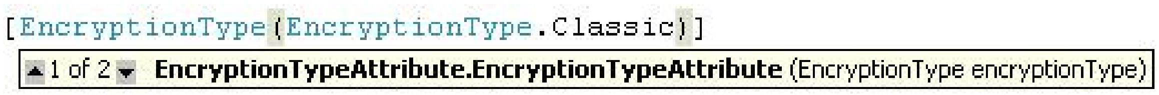
The last parameter tells CrypTool 2 the names of the provided icons. This parameter is an array composed of strings in the format *<assembly name>/<file name>* or *<assembly name>/<path>/<file name>*.

The first and most important icon is the plugin icon, which will be shown in CrypTool 2 in the ribbon bar and navigation pane. Once the icon has been added to the project as described in Section 2.5, we must accordingly tell CrypTool 2 where to find the icon. This can be seen above in Figure 2.18.

If your plugin will use additional icons, you should define the paths to each of them by adding the path strings to the *[PluginInfo]* attribute parameter list, each separated by a comma. We have added two further icons for the context menu in the CrypTool 2 workspace. (If you do choose to add more icons, don't forget to add the icons to your solution.)

2.6.3 The *[EncryptionType]* attribute

The third and last attribute, *[EncryptionType]*, is needed to tell CrypTool 2 what type of plugin we are creating. CrypTool 2 uses this information to place the plugin in the correct group in the navigation pane and ribbon bar. In our example, since Caesar is a classical algorithm, we will define the follows:



```
[EncryptionType(EncryptionType.Classic)]
```

1 of 2 **EncryptionTypeAttribute.EncryptionTypeAttribute** (EncryptionType encryptionType)

Figure 2.21: A fully-defined *[EncryptionType]* attribute.

The possible values of the *[EncryptionType]* attribute are as follows:

- Asymmetric — for asymmetrical encryption algorithms, such as RSA.
- Classic — for classical encryption or hash algorithms, such as Caesar or MD5.
- Hybrid — for algorithms which are actually a combination of several algorithms, such as algorithms in which the data is encrypted symmetrically and the encryption key asymmetrically.
- SymmetricBlock — for block cipher algorithms, such as DES, AES and Twofish.
- SymmetricStream — for stream cipher algorithms like RC4, Rabbit and SEAL.

2.7 Defining the private variables of the settings in the Caesar class

The next step is to define some private variables that are needed for the settings, input, and output data. In our example, this will look like the following:

```
public class Caesar : IEncryption
{
  #region Private variables
  private CaesarSettings settings;
  private string inputString;
  private string outputString;
  private enum CaesarMode { encrypt, decrypt };
  private List<CryptoolStream> listCryptoolStreamsOut = new List<
     CryptoolStream>();
  #endregion
```

If your algorithm works with longs strings of code, it is recommended to use the "CryptoolStream" data type. This was designed for input and output between plugins and to handle large amounts of data. To use the native CrypTool stream type, include the namespace "Cryptool.PluginBase.IO" with a "using" statement as explained in section 2.4.3.

The following private variables will be used in our example:

- CaesarSettings settings — required to implement the IPlugin interface properly.
- string inputString — string from which to read the input data.
- string outputString — string to which to save the output data.
- enum CaesarMode — used to select either encryption or decryption.
- List<CryptoolStream> listCryptoolStreamsOut — a list of all streams created by the plugin, which helps to perform a clean dispose.

2.8 Implementing the interfaces in the Caesar class

The next major step is to write out our implementations of the interfaces. First we will add a constructor to our class. We will use this to create an instance of our settings class and a function to handle events:

```
public class Caesar : IEncryption
{
  #region Private variables
  private CaesarSettings settings;
  private string inputString;
  private string outputString;
  private enum CaesarMode { encrypt, decrypt };
  private List<CryptoolStream> listCryptoolStreamsOut = new List<
     CryptoolStream>();
  #endregion

  public Caesar()
  {
    this.settings = new CaesarSettings();
    this.settings.LogMessage += Caesar_LogMessage;
  }
```

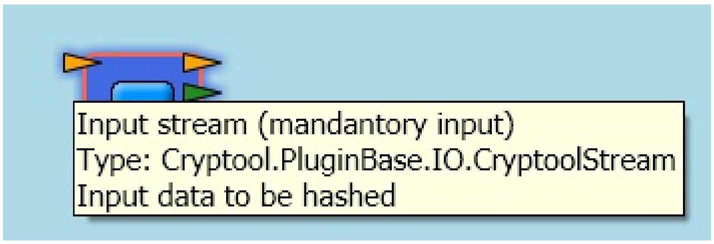
Secondly, we must implement the "Settings" property declared in the interface. An outline of this property should have been automatically generated by implementing the interface (see Section 2.4.4); just edit it appropriately to communicate with your settings class as we have done here:

```
public ISettings Settings
{
  get { return (ISettings)this.settings; }
  set { this.settings = (CaesarSettings)value; }
}
```

Thirdly, we must define five properties, each with an appropriate attribute. This step is necessary to tell CrypTool 2 if the properties are used for input our output and to provide the plugin with external data.

The attribute that we will use for each proprerty is called *[PropertyInfo]* and it 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. The possible values are:
  - Direction.Input
  - Direction.Output
- caption — the caption for the property displayed over the input of the icon after it has been placed in the editor), as seen below:

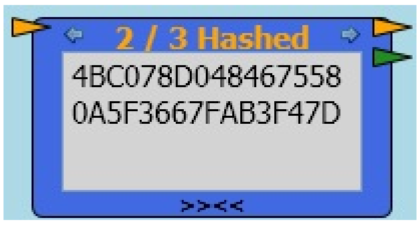


Input stream (mandantory input)
Type: Cryptool.PluginBase.IO.CryptoolStream
Input data to be hashed

Figure 2.22: A possible property caption and toolTip.

- toolTip — the toolTip for the property displayed over the input arrow of the icon after it has been placed in the editor, as seen above.
- descriptionUrl — currently not used; fill it with null or an empty string.
- mandatory — this flag determines whether an input must be attached by the user to use the plugin. If set to true, an input connection will be required or else the plugin will not be executed in the workflow chain. If set to false, connecting an input is optional. As this only applies to input properties, if the direction has been set to "output", this flag will be ignored.
- hasDefaultValue — if this flag is set to true, CrypTool 2 will assume that the property has a default input value that does not require user input.
- displayLevel — determines in which display levels your property will be shown in CrypTool 2. These are used to hide more advanced item from less-experienced users; a Beginner will not see the properties marked as any other level, but a Professional will have access to everything. The display levels are as follows:
  - DisplayLevel.Beginner
  - DisplayLevel.Experienced
  - DisplayLevel.Expert
  - DisplayLevel.Professional

- quickWatchFormat — determines how the content of the property will be shown in the quickwatch perspective. CrypTool accepts the following quickwatch formats:
  - QuickWatchFormat.Base64
  - QuickWatchFormat.Hex
  - QuickWatchFormat.None
  - QuickWatchFormat.Text



4BC078D048467558
0A5F3667FAB3F47D

Figure 2.23: A quickwatch display in hexadecimal.

- quickWatchConversionMethod — this is used to indicate a conversion method; most plugins do not ned to convert their data and thus should use a null value here. The quickwatch function uses the "default" system encoding to display data, so if your data is in another other format, such as Unicode or UTF-8, you should provide here the name of a conversion method as string. The method header for such a method should look something like the following:

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

The first of the five properties that we will define is "InputString". This is used to provide our plugin with the data to be encrypted or decrypted:

```
[PropertyInfo(Direction.InputData, "Text input", "Input a string to be
    processed by the Caesar cipher", "", true, false, DisplayLevel.
  Beginner, QuickWatchFormat.Text, null)]
public string InputString
{
  get { return this.inputString; }
  set
  {
    if (value != inputString)
    {
      this.inputString = value;
      OnPropertyChanged("InputString");
    }
  }
}
```

In the get method we simply return the value of the input data. The set method checks if the input value has changed, and, if so, sets the new input data and announces the change to the CrypTool 2 environment by calling the function "OnPropertyChanged(<*Property name*>)". This step is necessary for input properties to update the quickwatch view.

The output data property (which handles the input data after it has been encrypted or decrypted) will in our example look as follows:

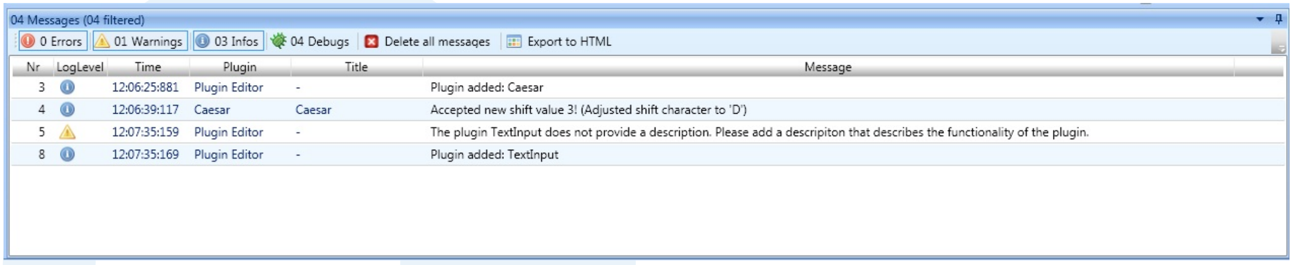
```
[PropertyInfo(Direction.OutputData, "Text output", "The string after
   processing with the Caesar cipher", "", false, false, DisplayLevel.
   Beginner, QuickWatchFormat.Text, null)]
public string OutputString
{
  get { return this.outputString; }
  set
  {
    outputString = value;
    OnPropertyChanged("OutputString");
  }
}
```

CrypTool 2 does not require implementing output set methods, as they will never be called from outside the plugin. Nevertheless, in our example the plugin accesses the property itself, and therefore we have chosen to implement the set method.

You can provide additional output data types if you so desire. In our example, we will also offer output data of type CryptoolStream, input data for external alphabets, and input data for the shift value of our Caesar algorithm. Note that for the first of these, the set method is not implemented since it will never be called. We shall define these properties as follows:

```
[PropertyInfo(Direction.OutputData, "propStreamOutputToolTip", "
   propStreamOutputDescription", "", false, false, DisplayLevel.
   Beginner, QuickWatchFormat.Text, null)]
public CryptoolStream OutputData
{
  get
  {
    if (outputString != null)
    {
      CryptoolStream cs = new CryptoolStream();
      listCryptoolStreamsOut.Add(cs);
      cs.OpenRead(Encoding.Default.GetBytes(outputString.ToCharArray()
         ));
      return cs;
    }
    else
    {
      return null;
    }
  }
  set { }
}

[PropertyInfo(Direction.InputData, "External alphabet input", "Input a
    string containing the alphabet to be used by Caesar.\nIf no
   alphabet is provided for this input, the internal default alphabet
   will be used.", "", false, false, DisplayLevel.Expert,
    QuickWatchFormat.Text, null)]
public string InputAlphabet
{
  get { return ((CaesarSettings)this.settings).AlphabetSymbols; }
  set
  {
    if (value != null && value != settings.AlphabetSymbols)
    {
      ((CaesarSettings)this.settings).AlphabetSymbols = value;
      OnPropertyChanged(''InputAlphabet'');
    }
  }
}

[PropertyInfo(Direction.InputData, "Shift value (integer)", "This is
    the same setting as the shift value in the Settings pane but as
    dynamic input.", "", false, false, DisplayLevel.Expert,
    QuickWatchFormat.Text, null)]
public int ShiftKey
{
  get { return settings.ShiftKey; }
  set
  {
    if (value != settings.ShiftKey)
    {
      settings.ShiftKey = value;
    }
  }
}
```

The CrypTool 2 API provides two methods to send messages from the plugin to the CrypTool 2 core: “GuiLogMessage” (used to send messages to the CrypTool 2 status bar) and “OnPropertyChanged” (used to inform the core of changes to the plugin data). The “GuiLogMessage” method is a nice mechanism to inform the user as to what your plugin is currently doing.



Nr	LogLevel	Time	Plugin	Title	Message
3	●	12:06:25:881	Plugin Editor	-	Plugin added: Caesar
4	●	12:06:39:117	Caesar	Caesar	Accepted new shift value 3! (Adjusted shift character to 'D')
5	⚠	12:07:35:159	Plugin Editor	-	The plugin TextInput does not provide a description. Please add a descripiton that describes the functionality of the plugin.
8	●	12:07:35:169	Plugin Editor	-	Plugin added: TextInput

Figure 2.24: An example status bar.

The method takes two parameters:

- Message — the text (of type string) to be shown in the status bar.
- NotificationLevel — the type of message, that is, its alert level:
  - NotificationLevel.Error
  - NotificationLevel.Warning
  - NotificationLevel.Info
  - NotificationLevel.Debug

Outlines of both of the related events will have been automatically generated by implementing the interface (see Section 2.4.4), but we must define appropriate methods as follows:

```
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));
}
```

Note that to use "PropertyChangedEventHandler" you must include the namespace "System. ComponentModel". Our collection of included namespaces should now look as follows:

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

using Cryptool.PluginBase;
using Cryptool.PluginBase.Cryptography;
using Cryptool.PluginBase.IO;
using Cryptool.PluginBase.Miscellaneous;
```

2.9 Completing the algorithmic code of the Caesar class

At this point, the plugin should be ready to be read by and shown correctly in the CrypTool 2 application. However, we haven't actually implemented the algorithm yet. This should be done in the "Execute()" function, as this is what CrypTool 2 will always call first. The actual functionality of your algorithm, as well as the structure thereof, is up to you. Note that an outline of the "Execute()" function will have been automatically generated by implementing the interface (see Section 2.4.4).

We have chosen to split our algorithm's encryption and decryption into two separate functions, which will both ultimately call the "ProcessCaesar()" function. Below is our implementation of the Execute() function:

```
private void ProcessCaesar(CaesarMode mode)
{
  CaesarSettings cfg = (CaesarSettings)this.settings;
  StringBuilder output = new StringBuilder("");
  string alphabet = cfg.AlphabetSymbols;

  // In case we are working in case-insensitive mode, we will use
  // only capital letters, hence we must transform the whole alphabet
  // to uppercase.
  if (!cfg.CaseSensitiveAlphabet)
  {
    alphabet = cfg.AlphabetSymbols.ToUpper();
  }

  if (inputString != null)
  {
    for (int i = 0; i < inputString.Length; i++)
    {
      // Get the plaintext char currently being processed.
      char currentchar = inputString[i];

      // Store whether it is upper case (otherwise lowercase is
         assumed).
      bool uppercase = char.IsUpper(currentchar);

      // Get the position of the plaintext character in the alphabet.
      int ppos = 0;
      if (cfg.CaseSensitiveAlphabet)
      {
        ppos = alphabet.IndexOf(currentchar);
      }
      else
      {
        ppos = alphabet.IndexOf(char.ToUpper(currentchar));
      }

      if (ppos >= 0)
      {
        // We found the plaintext character in the alphabet,
        // hence we will commence shifting.
        int cpos = 0;
        switch (mode)
        {
          case CaesarMode.encrypt:
            cpos = (ppos + cfg.ShiftKey) % alphabet.Length;
            break;
          case CaesarMode.decrypt:
            cpos = (ppos - cfg.ShiftKey + alphabet.Length) % alphabet.
               Length;
            break;
        }

        // We have the position of the ciphertext character,
        // hence just output it in the correct case.
        if (cfg.CaseSensitiveAlphabet)
        {
          output.Append(alphabet[cpos]);
        }
        else
        {
          if (uppercase)
          {
            output.Append(char.ToUpper(alphabet[cpos]));
          }
          else
          {
            output.Append(char.ToLower(alphabet[cpos]));
          }
        }
      }
      else
      {
        // The plaintext character was not found in the alphabet,
        // hence proceed with handling unknown characters.
        switch ((CaesarSettings.UnknownSymbolHandlingMode)cfg.
           UnknownSymbolHandling)
        {
          case CaesarSettings.UnknownSymbolHandlingMode.Ignore:
            output.Append(inputString[i]);
            break;
          case CaesarSettings.UnknownSymbolHandlingMode.Replace:
            output.Append('?');
            break;
        }
      }

      // Show the progress.
      if (OnPluginProgressChanged != null)
      {
        OnPluginProgressChanged(this, new PluginProgressEventArgs(i,
                inputString.Length - 1));
         }
      }
      outputString = output.ToString();
      OnPropertyChanged("OutputString");
      OnPropertyChanged("OutputData");
   }
}

public void Encrypt()
{
   ProcessCaesar(CaesarMode.encrypt);
}

public void Decrypt()
{
   ProcessCaesar(CaesarMode.decrypt);
}

public void Execute()
{
   switch (settings.Action)
   {
      case 0:
         Caesar_LogMessage("Encrypting", NotificationLevel.Debug);
         Encrypt();
         break;
      case 1:
         Caesar_LogMessage("Decrypting", NotificationLevel.Debug);
         Decrypt();
         break;
      default:
         break;
   }
}
```

It is important to make sure that all changes to the output properties will be announced to the CrypTool 2 environment. In our example this happens by calling the set method of OutputData, which in turn calls “OnPropertyChanged” for both output properties “OutputData” and “OutputDataStream”. Instead of calling the property’s set method you can instead call “OnPropertyChanged” directly within the “Execute()” method.

You have probably noticed that the “ProgressChanged” method is undefined. This can be used to show the current algorithm process as a progress bar in the plugin icon. To use this method and compile successfully, you must declare this method, as we have done for our example below:

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

2.10 Performing a clean dispose

Be sure you have closed and cleaned all your streams after execution before CrypTool 2 decides to dispose the plugin instance. Though not required, we will run the disposal code before execution as well. We will expand the associated automatically generated methods (see Section 2.4.4) as follows:

```
public void Dispose()
{
  foreach(CryptoolStream stream in listCryptoolStreamOut)
  {
    stream.Close();
  }
  listCryptoolStreamOut.Clear();
}

public void PostExecution()
{
  Dispose();
}

public void PreExecution()
{
  Dispose();
}
```

2.11 Finishing the implementation

When adding plugin instances to the CrypTool 2 workspace, the application core checks whether the plugin runs without any exceptions. If any method inherited from IPlugin throws an exception, CrypTool 2 will display an error message and prohibit use of the plugin. Therefore, we must remove the "NotImplementedException" from the automatically generated methods "Initialize()", "Pause()" and "Stop()". In our example it will be sufficient to provide empty implementations.

```
public void Initialize()
{
}

public void Pause()
{
}

public void Stop()
{
}
```

The methods "Presentation()" and "QuickWatchPresentation()" can be used to provide a specialized visualization of the plugin algorithm to 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 have chosen not to implement a custom visualization. Therefore we will simply return "null":

```
public UserControl Presentation
{
  get { return null; }
}

public UserControl QuickWatchPresentation
{
  get { return null; }
}
```

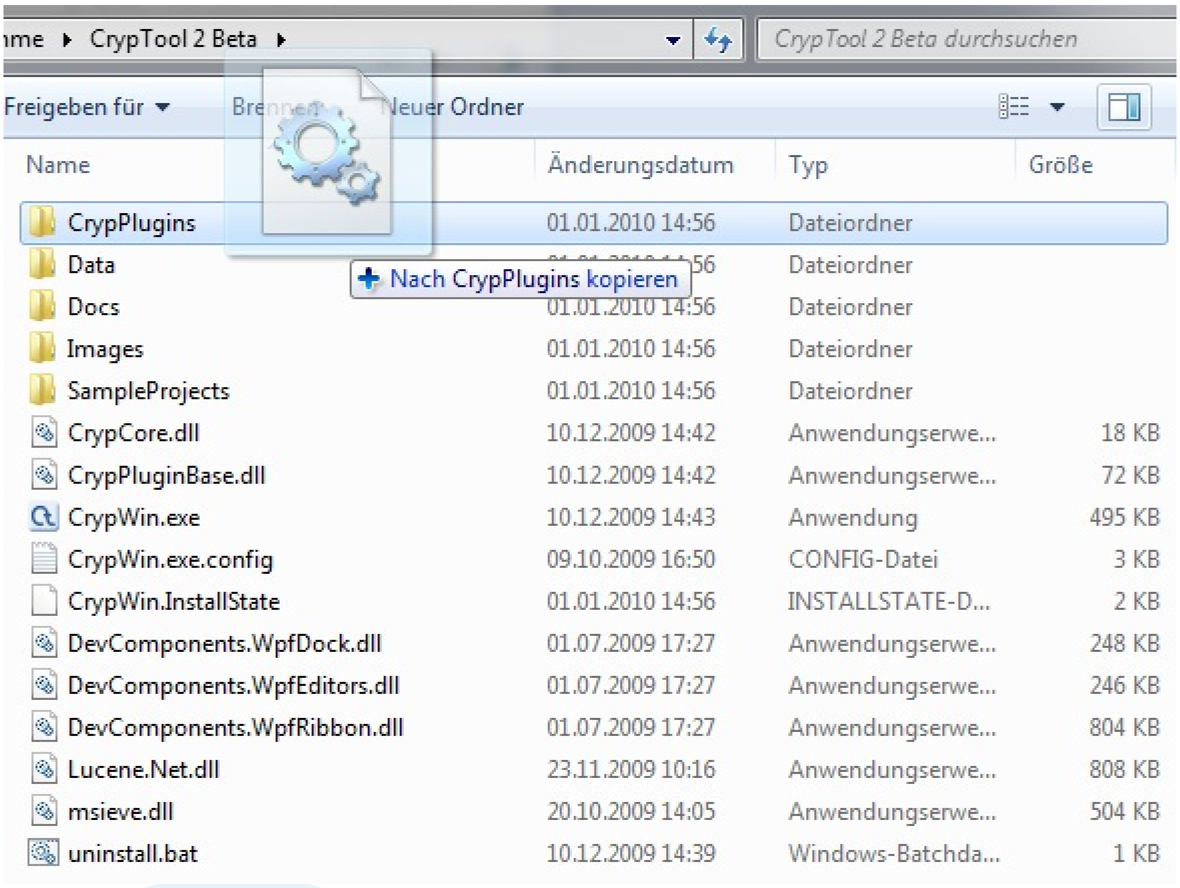
Your plugin should compile without errors at this point.

2.12 Importing and testing the plugin

After you have built the plugin, you need to move the newly created plugin DLL to a location where CrypTool 2 can find it. There are a few different ways to accomplish this. You can find the DLL file in *\CrypPluginBase\bin\Debug*.

2.12.1 Global storage

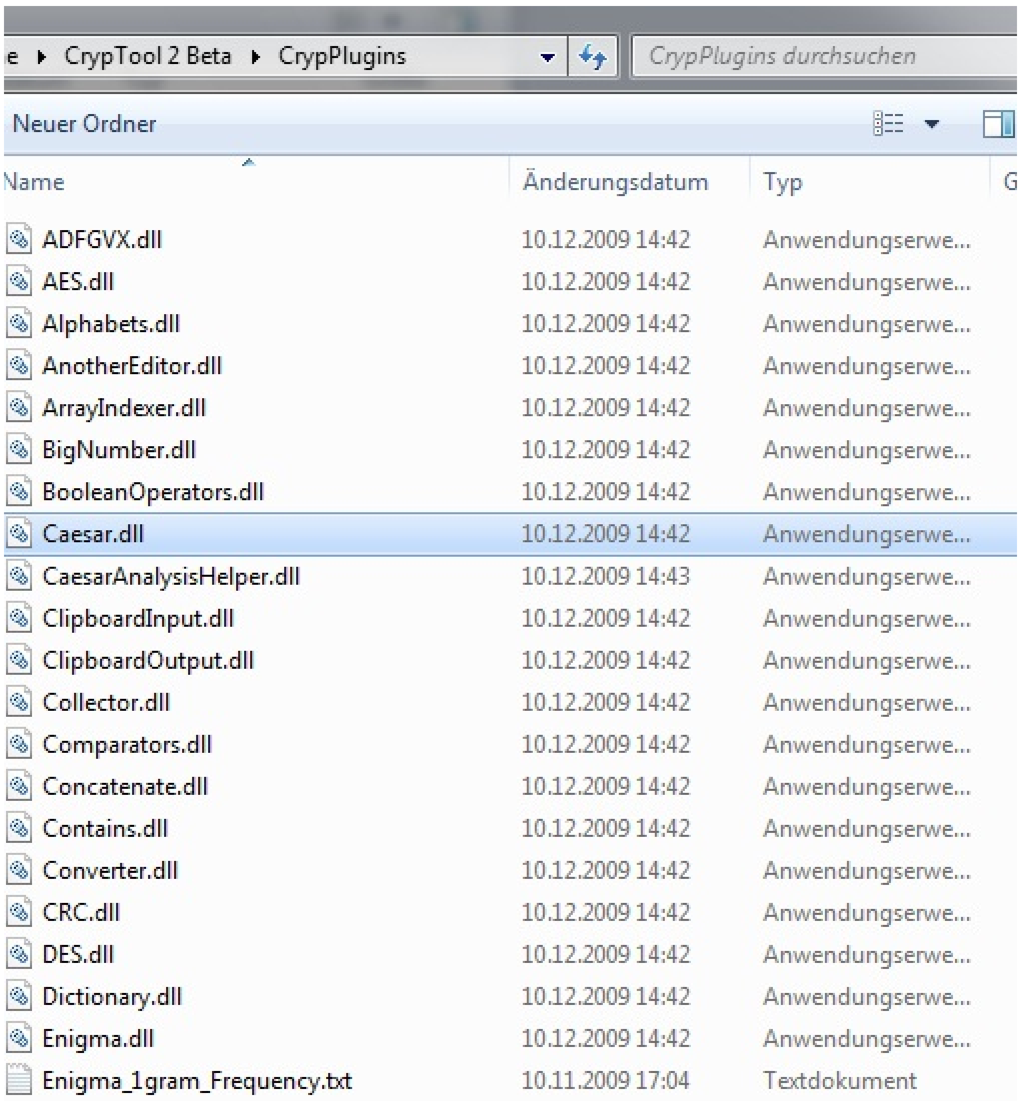
The first option is to copy your plugin's DLL file to the "CrypPlugins" folder in which the CrypTool 2 executable ("CrypWin.exe") can be found.



Name	Änderungsdatum	Typ	Größe
CrypPlugins	01.01.2010 14:56	Dateiordner	
Data	01.01.2010 14:56	Dateiordner	
Docs	01.01.2010 14:56	Dateiordner	
Images	01.01.2010 14:56	Dateiordner	
SampleProjects	01.01.2010 14:56	Dateiordner	
CrypCore.dll	10.12.2009 14:42	Anwendungserwe...	18 KB
CrypPluginBase.dll	10.12.2009 14:42	Anwendungserwe...	72 KB
CrypWin.exe	10.12.2009 14:43	Anwendung	495 KB
CrypWin.exe.config	09.10.2009 16:50	CONFIG-Datei	3 KB
CrypWin.InstallState	01.01.2010 14:56	INSTALLSTATE-D...	2 KB
DevComponents.WpfDock.dll	01.07.2009 17:27	Anwendungserwe...	248 KB
DevComponents.WpfEditors.dll	01.07.2009 17:27	Anwendungserwe...	246 KB
DevComponents.WpfRibbon.dll	01.07.2009 17:27	Anwendungserwe...	804 KB
Lucene.Net.dll	23.11.2009 10:16	Anwendungserwe...	808 KB
msieve.dll	20.10.2009 14:05	Anwendungserwe...	504 KB
uninstall.bat	10.12.2009 14:39	Windows-Batchda...	1 KB

Figure 2.25: Copying the plugin to the global storage folder

This folder is known as "global storage" in the CrypTool 2 architecture. Changes in this folder will affect all users on a multi-user Windows platform. You should now restart CrypTool 2.

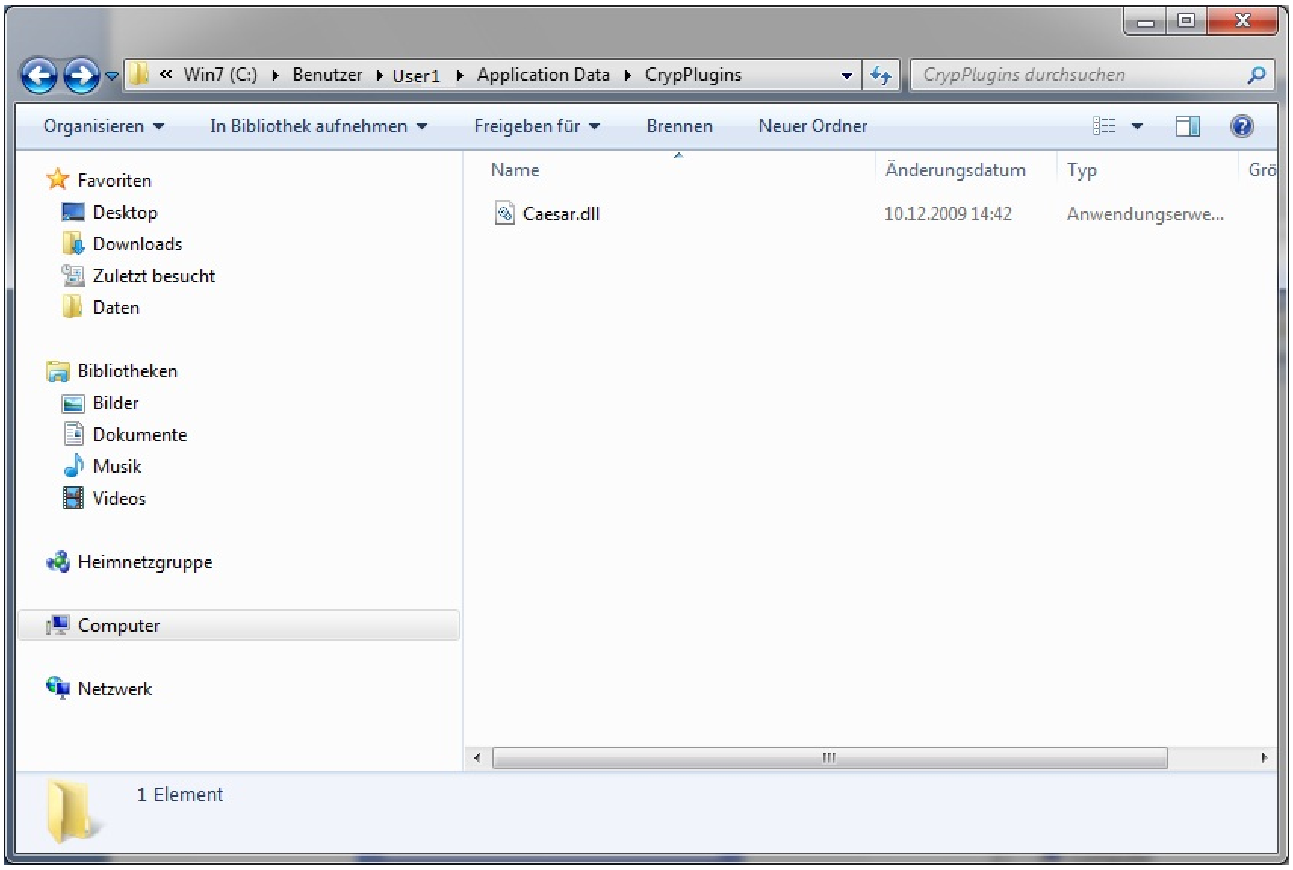


Name	Änderungsdatum	Typ	G
ADFGVX.dll	10.12.2009 14:42	Anwendungserwe...	
AES.dll	10.12.2009 14:42	Anwendungserwe...	
Alphabets.dll	10.12.2009 14:42	Anwendungserwe...	
AnotherEditor.dll	10.12.2009 14:42	Anwendungserwe...	
ArrayIndexer.dll	10.12.2009 14:42	Anwendungserwe...	
BigNumber.dll	10.12.2009 14:42	Anwendungserwe...	
BooleanOperators.dll	10.12.2009 14:42	Anwendungserwe...	
Caesar.dll	10.12.2009 14:42	Anwendungserwe...	
CaesarAnalysisHelper.dll	10.12.2009 14:43	Anwendungserwe...	
ClipboardInput.dll	10.12.2009 14:42	Anwendungserwe...	
ClipboardOutput.dll	10.12.2009 14:42	Anwendungserwe...	
Collector.dll	10.12.2009 14:42	Anwendungserwe...	
Comparators.dll	10.12.2009 14:42	Anwendungserwe...	
Concatenate.dll	10.12.2009 14:42	Anwendungserwe...	
Contains.dll	10.12.2009 14:42	Anwendungserwe...	
Converter.dll	10.12.2009 14:42	Anwendungserwe...	
CRC.dll	10.12.2009 14:42	Anwendungserwe...	
DES.dll	10.12.2009 14:42	Anwendungserwe...	
Dictionary.dll	10.12.2009 14:42	Anwendungserwe...	
Enigma.dll	10.12.2009 14:42	Anwendungserwe...	
Enigma_1gram_Frequency.txt	10.11.2009 17:04	Textdokument	

Figure 2.26: Inside the CrypPlugins folder (the global storage).

2.12.2 Custom storage

The second possibility is to copy your plugin's DLL file to the "CrypPlugins" folder located in the "Application Data" folder in your home folder. In Windows XP, the home folder path should be as follows: *C:\Documents and Settings\<user name>\Application Data\CrypPlugins*, and in Vista and Windows 7 the path should look like: *C:\Users\<user name>\Application Data\CrypPlugins*. This home folder path is called "custom storage" in the CrypTool architecture. Changes in this folder will only take effect for current user. After copying the file, you must restart CrypTool 2.



Name	Änderungsdatum	Typ	Grö
Caesar.dll	10.12.2009 14:42	Anwendungserwe...	

Figure 2.27: The custom storage folder.

2.12.3 Importing directly

Alternatively, you can import new plugins directly from the CrypTool 2 interface. Just run CrypWin.exe and select the "Download Plugins" button. An "Open File Dialog" window will open and ask where the new plugin is located. After selecting the new plugin, CrypTool 2 will automatically import the plugin to the custom storage folder. With this option you will not have to restart the program. All corresponding menu entries will be updated automatically. Note that this import function only accepts **signed** plugins, and also that this option is just a temporary solution: in the future this will be done online by a web service.

2.12.4 Using build settings

Yet another option is to use the build settings in your plugin's project properties to copy the DLL automatically after building it in Visual Studio. To set this up, right-click on your plugin project and select "Properties":

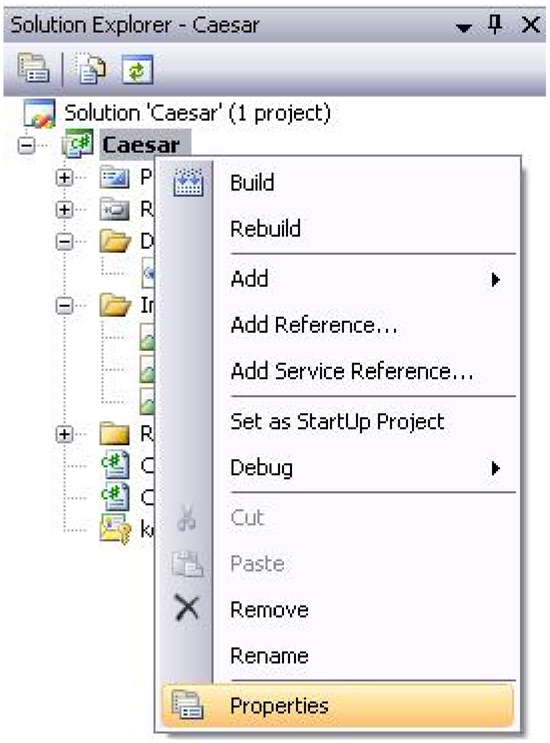
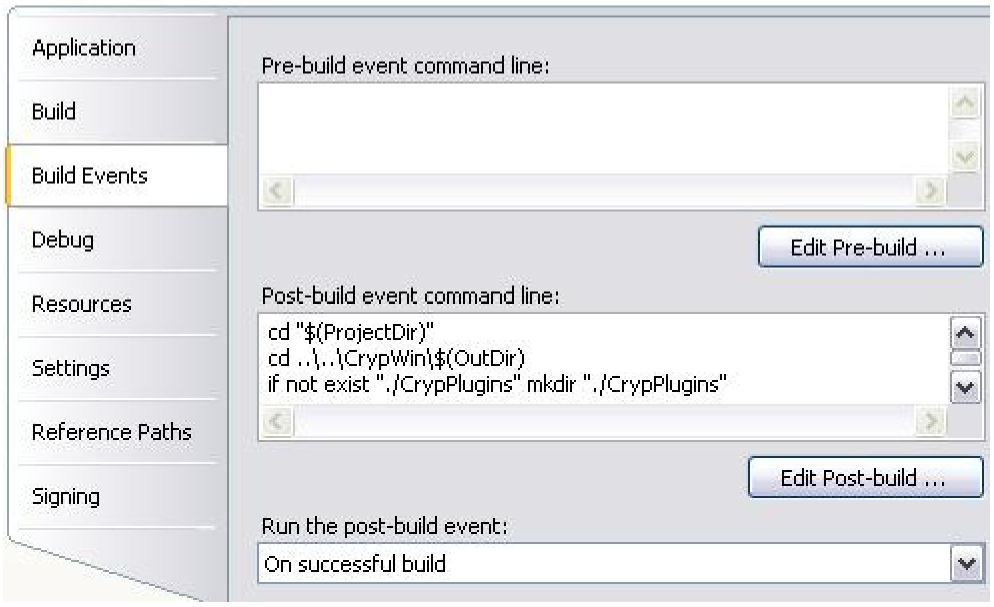


Figure 2.28: Selecting the solution properties.

Then select “Build Events”:



```
cd "$(ProjectDir)"
cd ..\..\CrypWin\$(OutDir)
if not exist "./CrypPlugins" mkdir "./CrypPlugins"
```

On successful build

Figure 2.29: Setting the build events.

And finally, enter the following text 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 will need to change the marked fields to your particular plugin’s name.

2.13 Downloading the example and template

If you didn’t download the entire CrypTool 2 source code as described in Section 1.2.1, but you want a copy of the source code for the Caesar algorithm that was used as an example in this guide, you can download it as a Visual Studio **solution** from the following location:

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

We have also created a Visual Studio plugin template to help with the development of new plug-ins. This can be found here:

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

2.14 Drawing the workflow of your plugin

Each plugin should have an associated workflow file to show the algorithm in action in CrypTool 2. Such a file can be automatically created by simply saving a CrypTool 2 workspace project featuring your plugin. Below is a possible workflow for our Caesar example:

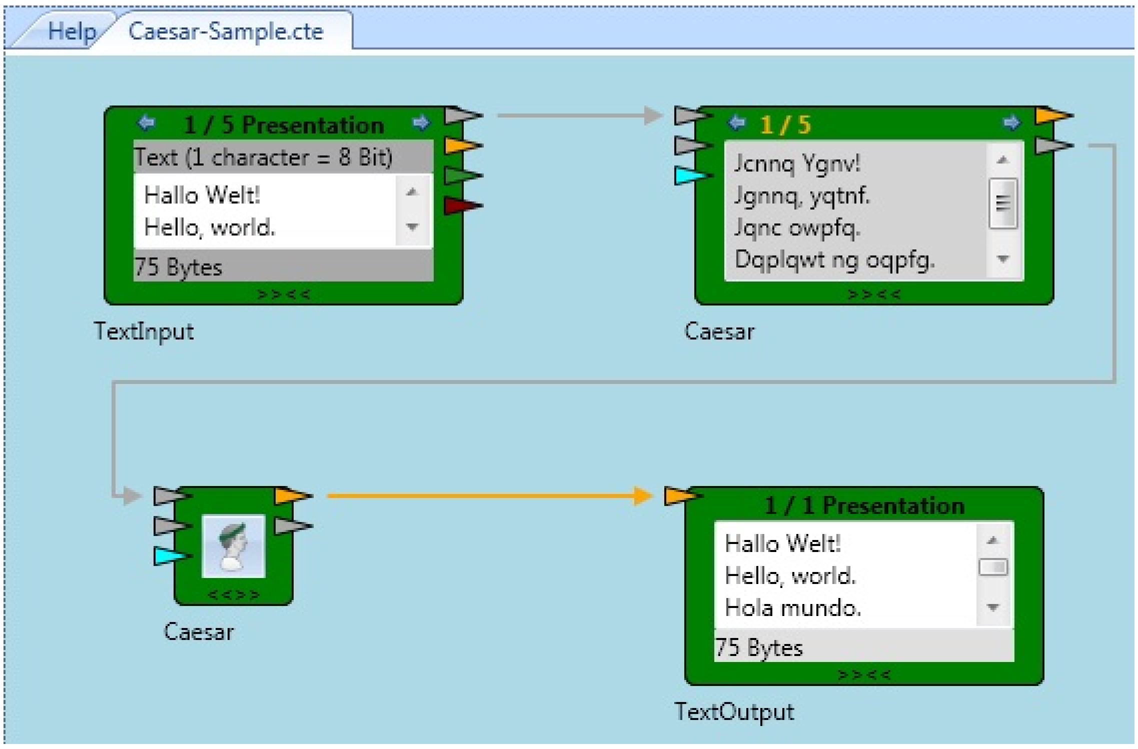


Figure 2.30: A sample workflow diagram for the Caesar algorithm.