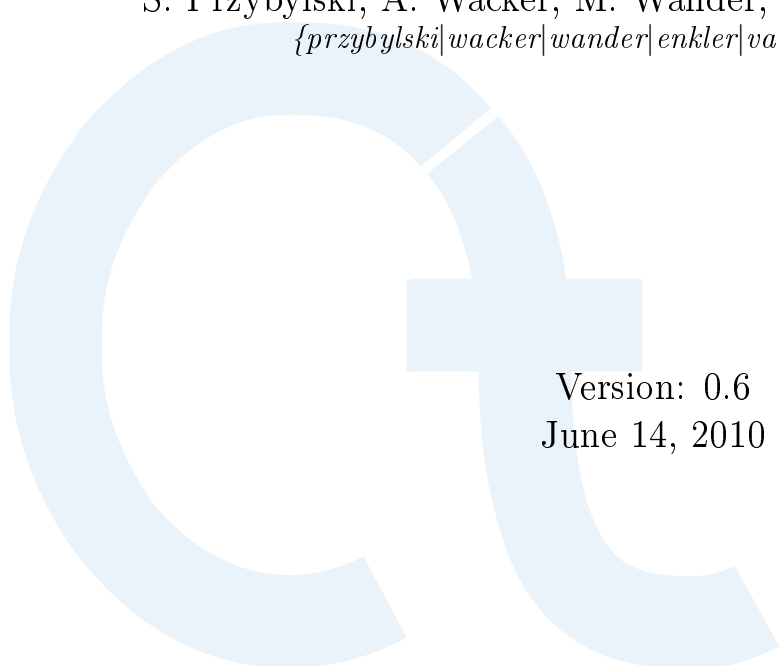CrypTool 2.0

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: 0.6

June 14, 2010

CrypTool 2 is the modern successor of the well-known e-learning platform for cryptography and cryptanalysis CrypTool 1, which is used worldwide for educational purposes at schools and universities as well as 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 reach these goals, 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 2010 (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 CrypTool 2. As of January 2010, the program consists of over 7000 lines of C# code in the core application and over 250,000 lines of C# code in about 100 plugins.

For further information, please visit the CrypTool 2 website at http://www.cryptool2.vs.uni-due.de.

Contents

List of Figures

1. Developer Guidelines

CrypTool 2.0 is built upon state-of-the-art technologies such as .NET 4.0 and the Windows Presentation Foundation (WPF). Before you can start writing code and adding to the development of the project, a few things need to be considered. To make this process easier, please read through this document and follow the instructions closely. This document exists to help get you started by showing you how CrypTool 2 plugins are built in order to succesfully interact with the application core. We have tried to be very thorough, but if you encounter a problem or error that is not described here, please let us know. Not only do we want to help get you up and running, but we also want to add the appropriate information to this guide for the benefit of other future developers.

In this first chapter we will describe all steps necessary in order to compile CrypTool 2 on your own computer. 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 latest version of the source code

1.1. Prerequisites

Since CrypTool 2 is based on Microsoft .NET 4.0, you will need a Microsoft Windows environment. (Currently no plans exist for porting this project to Mono or 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 2010** or **Microsoft Visual C# 2010 Express**. Make sure to always install the latest service packs for Visual Studio.

In order to run or compile our source code you will need at least the **Microsoft .NET 4.0**. Usually the installation of Visual Studio also installs the .NET framework, but if you do not have the latest version, you can get it for free from Microsoft's website. Once the framework has been installed, your development environment should be ready for our source code.

1.2. Accessing the Subversion (SVN) repository

Next you will need a way of accessing and downloading the source code. For the CrypTool 2 project we use **Subversion (SVN)** for version control, and hence you will need an SVN client, i.e. **TortoiseSVN**, **AnkhSVN** or the **svn commandline from cygwin**, to access our repository. It does not matter which client you use, but if SVN is new to you, we suggest using TortoiseSVN, since it offers a handy, straightforward Windows Explorer integration. We will guide you through how to use TortoiseSVN, although you should be able to use any SVN client in a similar fashion.

1.2.1. Checking out the sources

First, download and install TortoiseSVN. This will require you to reboot your computer, but once it is back up and running, create a directory (for instance, *CrypTool2*) somewhere on your computer for storing the local working files. Right-click on this directory; now that TortoiseSVN has been installed, you should see a few new items in the context menu (Figure 1.1). Select *SVN Checkout*.
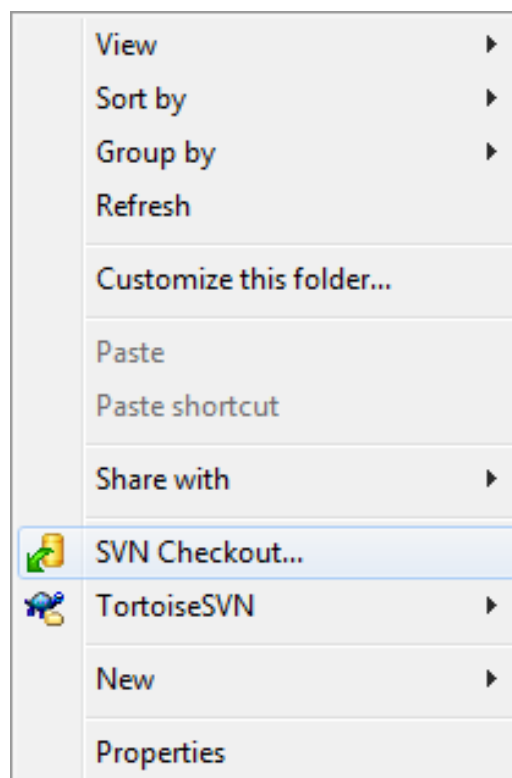

Figure 1.1.: Selecting *SVN Checkout* from the context menu after installing TortoiseSVN.

A window will now appear that will ask you for the URL of the repository that you would like to access. Our code repository is stored at `https://www.cryptool.org/svn/CrypTool2/trunk/`, and this is what you should enter in the appropriate field. The *Checkout directory* should already be filled in correctly with your new folder, and you shouldn't need to change any other options.
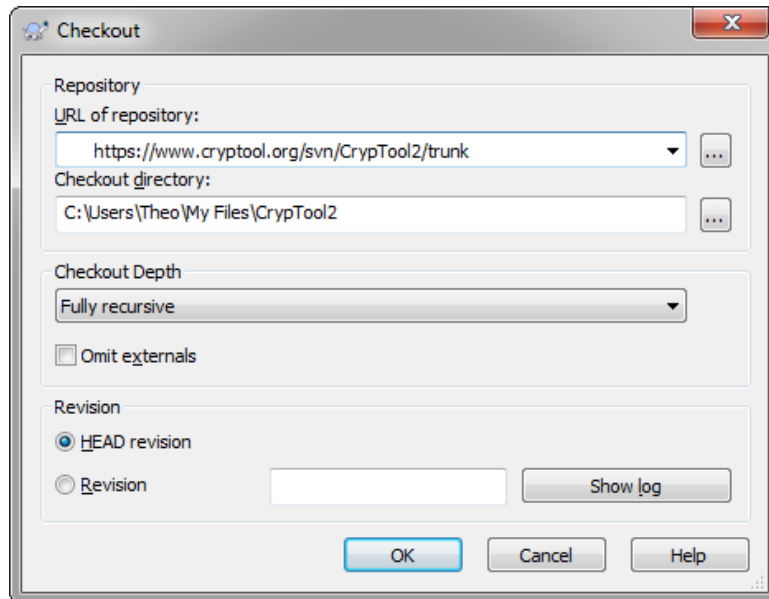
Figure 1.2.: Checking out the CrypTool 2 repository.

Then just hit *OK*. You may be asked to accept a certificate (which you should accept), and you will certainly be asked for login information. If you are a registered developer, you should have already been given a username and password, and you should enter them here. (These are the same username and password that you can use for the CrypTool 2 development wiki.) If you are a guest and only need read-only access, you can use "anonymous" as the username and an empty password. Mark the checkbox for saving your credentials if you don't want to enter them every time you work with the repository. (Your password will be saved on your computer.) Finally, hit *OK*, and the whole CrypTool 2 repository will begin downloading into your chosen local directory.

Since CrypTool 2 is a collaborative project with many developers, changes are made to the repository rather frequently. You should maintain a current working copy of the files to ensure your interoperability with the rest of the project, and thus you should update to the latest version as often as possible. You can do this by right-clicking on any directory within the working files and choosing *SVN Update* from the context menu.

A TortoiseSVN tutorial can be found at `http://www.mind.ilstu.edu/research/robots/iris5/developers/documentation/svntutorial/`.

1.2.2. Adjusting the SVN settings

If you are a registered developer, you can commit your file changes to the public CrypTool 2 repository. However, before you do, you should edit your settings to make sure that you only check in proper source code. First, bring up the TortoiseSVN settings window:
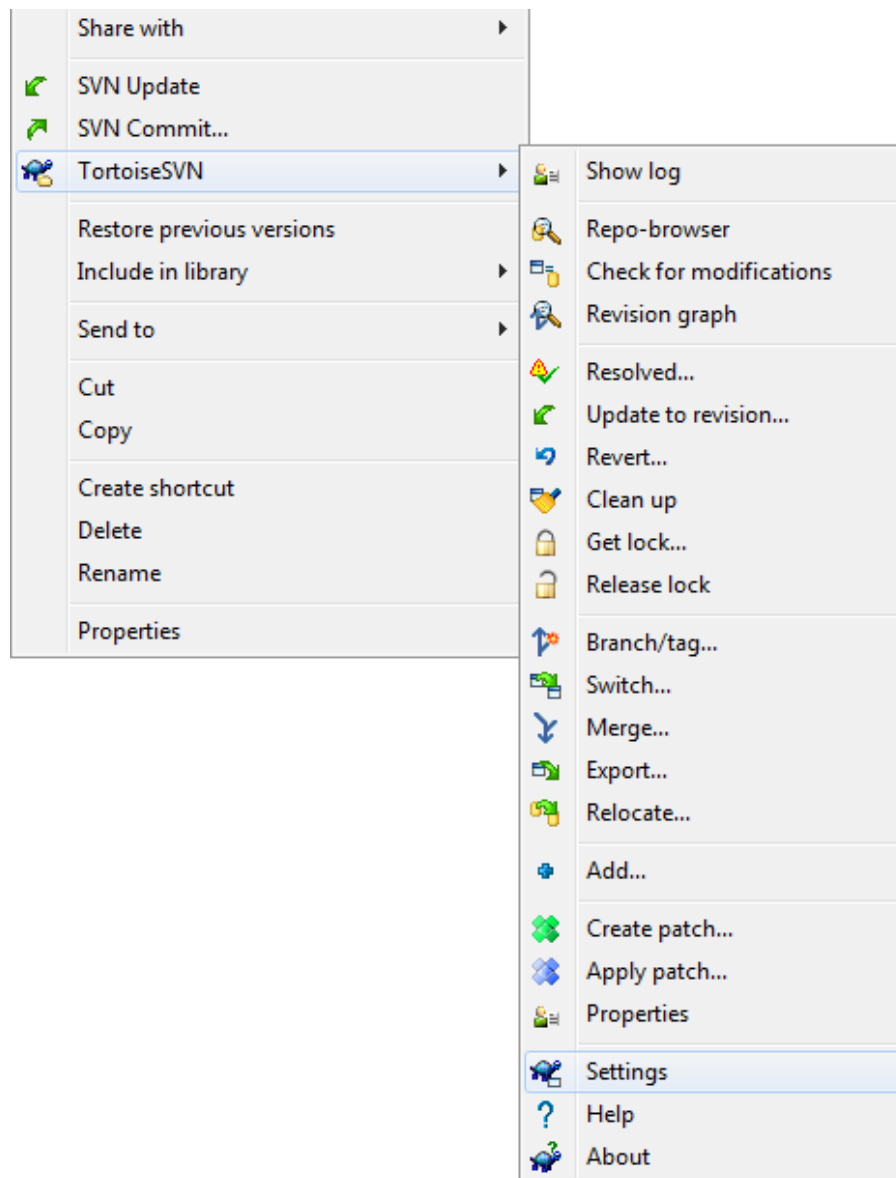
Figure 1.3.: Getting to the TortoiseSVN settings.

The settings window will look something like this:
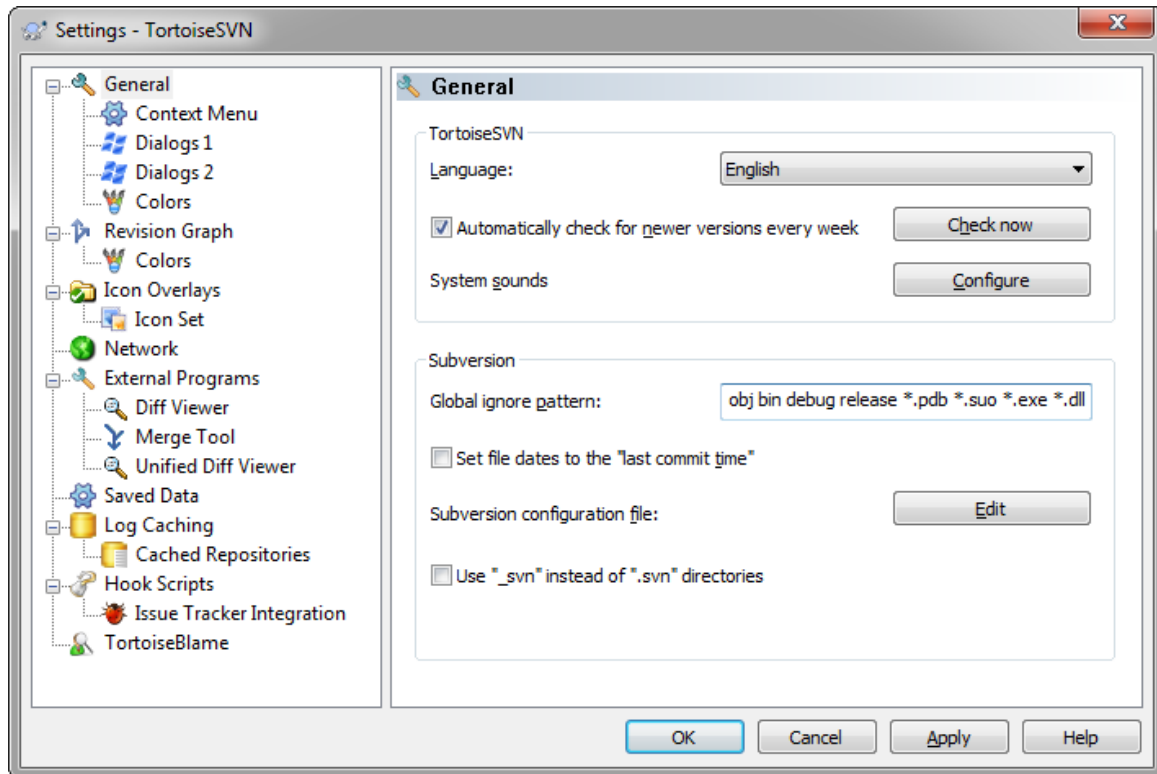
Figure 1.4.: The TortoiseSVN settings window with the proper ignore pattern.

Then in the *Global ignore pattern* field, please enter the following text:

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

   You are free to also leave in any default pattern text or to write your own additions; this pattern serves simply to tell TortoiseSVN what kinds of files to ignore. You can now click *OK* to save your settings and close the window.

1.2.3. Committing your changes

Once you start writing code and developing your plugin, you should check your work into the project repository. If you are reading this document in sequence, you are probably not ready to do this, but while we are on the topic of SVN we will describe the process. To upload your changes, right-click on a directory within the working files that contains your changes and select *SVN Commit* from the context menu:
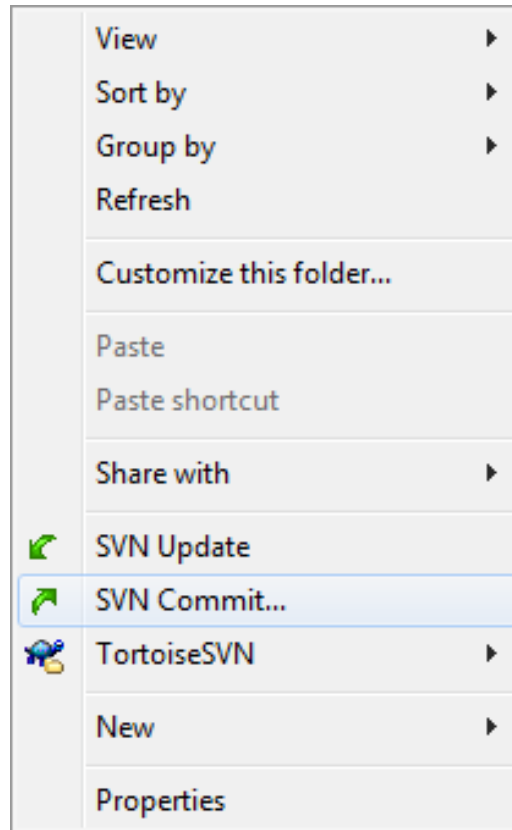
Figure 1.5.: Selecting *SVN Commit* from the context menu.

When you commit your code, you must enter a comment to describe what you have changed. *Meaningful descriptions* will help other developers comprehend your updates. You can also select exactly which files you want to check in. The ignore pattern that we recommended should prevent most undesirable files from being included, but double-check to make sure everything you want to upload is included and nothing more. In general, you should never check in compiled or automatically generated files. For example, do not check in the entire `bin\` and `obj\` directories that Visual Studio generates. The server will reject your commits if you try to do so. You should commit your sources to our SVN repository as often as you can, even if your work is unfinished or there are bugs. However, your committed code should not break any part of the existing project, so please make sure the public solution still compiles and runs successfully.
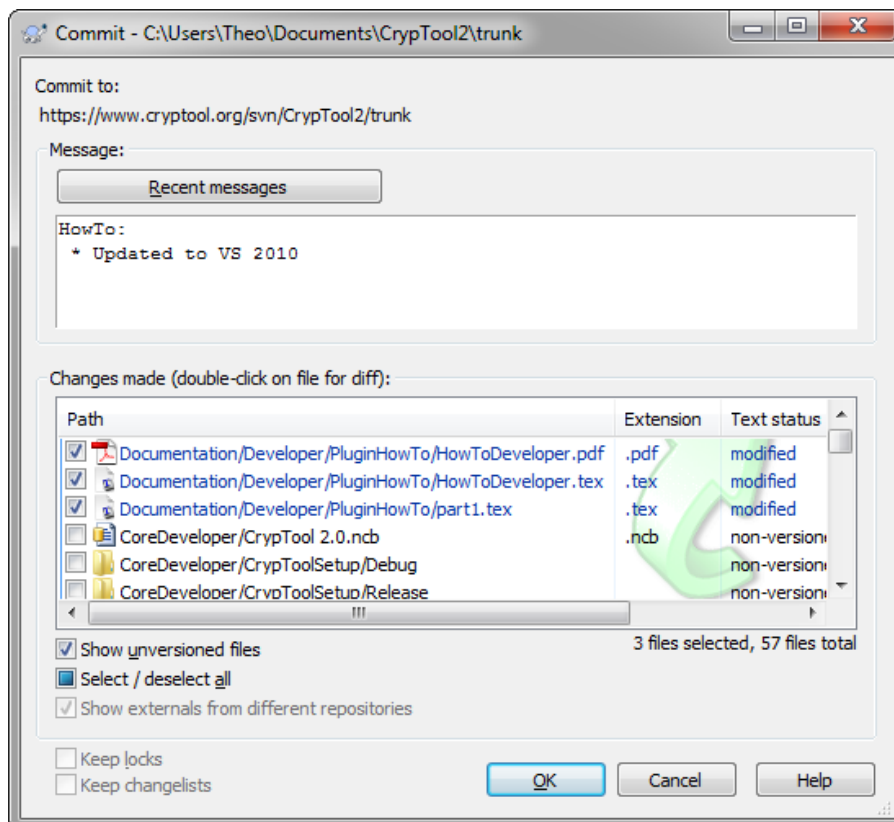
Figure 1.6.: Providing comments for a commit.

You can use the SVN comments to link to your changes to a particular issue or bug ticket on the CrypTool 2 development wiki. (The list of active tickets can be found here.) The following commands are supported (note that there are multiple variations of each command that are functionally identical):

```
closes, fixes:
```
The specified ticket will be closed and the contents of this commit message will be added to its notes.

```
references, refs, addresses, re:
```
The contents of this commit message will be added to the specified ticket's notes, but the status will be left unaltered.

You can apply the commands to multiple tickets simultaneously. The command syntax is as follows (again note that there are multiple variations that are functionally identical):

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

You can also use more than one command in a message if necessary. For example, if you want to close tickets #10 and #12, and add a note to #17, you could type the following:

```
Changed blah and foo to do this or that. Fixes #10 and #12, and refs #17.
```

The comments can also be used to override the ignore pattern that the server is designed to block. However, please do not do this unless you are absolutely sure that you know what you are doing. If you are, you must use the *override-bad-extension* command and provide an explicit list of the file and directory names that you want to upload that need to override the ignore pattern. For example, if you want to check in a library file named *someLib.dll*, you must write something like the following:

```
This library is referenced by project xy.

override-bad-extension: someLib.dll
```

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 with Visual Studio 2010

By this point you should have checked out a copy of the entire CrypTool 2 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 plugin components nicely arranged. If 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 ***CrypStartup*** there and make sure it is selected as startup project (right-click on it and select *Set as StartUp Project* from the context menu). Next, go to the menu bar and make sure the target platform is set correctly (Figure 1.7). If your operating system is a 32 bit installation, you have to select **x86**. If you have a 64 bit operating system, you may use one of both, x64 or x86. If in doubt, select x86. Then click *Build* → *Build Solution* in the menubar to start the build process.
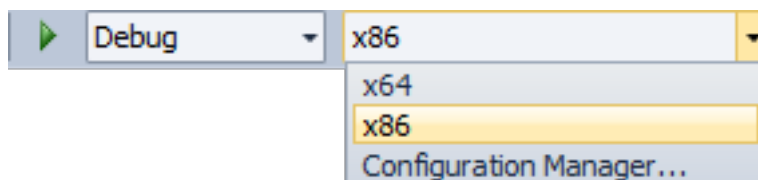

Figure 1.7.: Selecting x86 as target platform.

You may have to wait a while for the program to compile. Once it is finished, select *Debug → Start Debugging*. CrypTool 2 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 CrypWin and AnotherEditor, you can use the ***CrypTool 2.0.sln*** solution from the `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 *CrypWin.exe*** that is visible to everybody else. Thus, when committing to the repository, please make sure you *really* want to check in a new binary.

1.4. Compiling the sources with Visual C# 2010 Express

With Visual C# Express the build process is basically the same as with Visual Studio. When opening the solution file, you will receive two error messages. The first is because Visual C# does not support solution folders and shows all plugin projects as a flat list in the solution explorer. However, this is merely a visual defect. The second error message occurs because Visual C# does not support unit tests and thus is not able to load the project *DevTestMethods*. Again, this does not interfere with opening, writing, compiling, running, or debugging any other plugins.

1.5. Downloading the plugin template

Before you can start implementing a new plugin, you will need to download the CrypTool 2 plugin template. The template is located at the CrypTool 2 website at `http://www.cryptool2.vs.uni-due.de/index.php?page=33&lm=3`. Save the template zip file in your documents folder in the subdirectory `Visual Studio 2010\Templates\ProjectTemplates\`. (This applies to both Visual Studio and Visual C# Express.) Do not unpack the zip file.
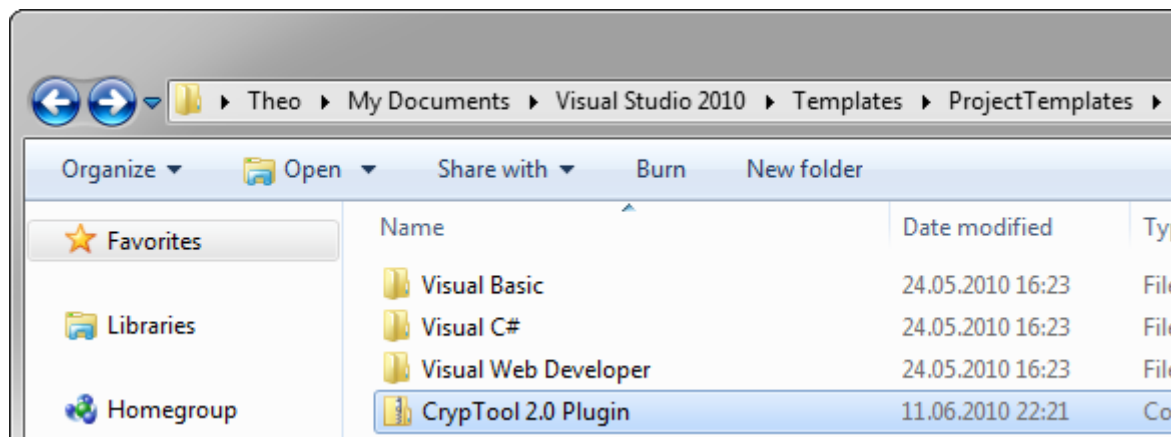
Figure 1.8.: Saving plugin template.

2. Plugin Implementation

In this chapter we provide step-by-step instructions for implementing your own CrypTool 2 plugin. We shall assume that you have already retrieved the CrypTool 2 source code from the SVN repository, set up Visual Studio 2010 or Visual C# 2010 Express to build CrypTool 2, and you have placed the plugin template in the right place.

2.1. Downloading the example

We will use the **Caesar cipher** (also known as the **shift cipher**) as an example throughout this chapter. If you have not downloaded the entire CrypTool 2 source code as described in Section 1.2.1, you can also get a copy of just the source code for the Caesar algorithm referenced throughout this guide from the following location:

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

2.2. Creating a new project

Open the CrypTool 2 solution, right click in the solution explorer on *CrypPlugins* (or on the top solution item, if you're using Visual C# Express) and select *Add → New Project*. In the dialog window, select *Visual C# → CrypTool 2.0 Plugin* as project template and enter a unique name for your new plugin project (such as *Caesar* in our case). The **next step is crucial** to ensure that your plugin will compile and run correctly: select the subdirectory `CrypPlugins\` as the location of your project (Figure 2.1).
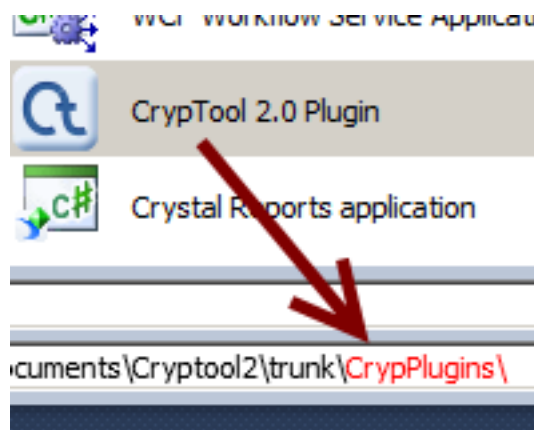
Figure 2.1.: Creating a new CrypTool 2 plugin project.

As the project basics are already correctly set up in the template, you should now be able to compile the new plugin. First of all, rename the two files in the solution explorer to something more meaningful, for example `Caesar.cs` and `CaesarSettings.cs`.
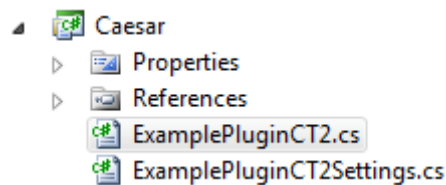
Figure 2.2.: Items of a new plugin project.

2.3. Adapting the plugin skeleton

If you look into the two C# source files (Figure 2.2), you will notice a lot of comments marked with the keyword `HOWTO`. These are hints as to what you should change in order to adapt the plugin skeleton to your custom implementation. Most `HOWTO` hints are self-explanatory and thus we won't discuss them all in detail. For example, at the top of both source files, there is a hint to enter your name and affiliation in the license boilerplate:

```
1 /* HOWTO: Change year, author name and organization.
2    Copyright 2010 Your Name, University of Duckburg
```

After you have done this, remove the `HOWTO` hint:

```
1 /*
2    Copyright 2010 John Doe, University of Duisburg-Essen
```
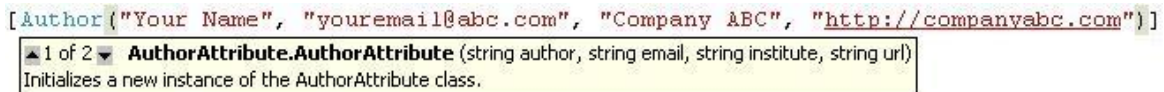
2.4. Defining the attributes of the Caesar class

The next thing we will do in our example *Caesar.cs* is define the attributes of our class. These attributes are used to provide necessary information for the CrypTool 2 environment. If they are not properly defined, your plugin won't show up in the application user interface, 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 2 provides a set of custom attributes that are used to mark the different parts of your plugin. These attributes should be defined right before the class declaration.

2.4.1. The *[Author]* attribute

The *[Author]* attribute is optional, meaning that we are not required to define it. The attribute can be used to provide additional information about the plugin developer (or developers, as the case may be). This information will appear in the TaskPane, as for example in Appendix A. We will define the attribute to demonstrate how it should look in case you want to use it in your plugin.

Figure 2.3.: The defintion for the *[Author]* attribute.
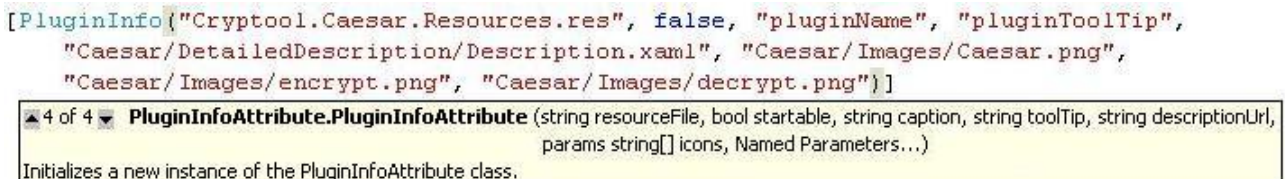
As can be seen above, the author attribute takes four elements of type string. These elements are:

- *Author* — the name of the plugin developer.

- *Email* — the email address of the plugin developer, should he or she wish to be available for contact.

- *Institute* — the organization, company or university with which the developer is affiliated.

- *URL* — the website of the developer or of his or her institution.

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

2.4.2. The *[PluginInfo]* attribute

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

Figure 2.4.: The defintion for the *[PluginInfo]* attribute.

This attribute has the following parameters:

- *Resource File* — the relative path of the associated resource file (if the plugin makes use of one). These files are used primarily to provide multilingual support for the plugin, although this is currently a work in progress. This element is optional.

- *Startable* — a flag that should be set to `true` only if the plugin is an input generator (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 using a resource file, the name of the field in the file with the caption data. This element is mandatory.

- *ToolTip* — a description of the plugin, or, if using a resource file, the name of the field in the resource file with the toolTip data. This element is optional.

- *DescriptionURL* — the local path of the description file (e.g. XAML file). This element is optional.

- *Icons* — an array of strings to define all the paths of the icons used in the plugin (i.e. the plugin icon described in Section 2.6). 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. First, it is possible to use the plugin without setting a caption, although this is not recommended, and future versions of the plugin will fail to load without a caption. Second, a zero-length toolTip string currently causes the toolTip to appear as an empty box in the application. Third, the toolTip and description do not currently support internationalization and localization. Since the precise formulation and functionality of this attribute is still being developed, it is recommended to view other plugins for examples.

In our example, the *resourceFile* parameter is 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.

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 caption and toolTip. (We could also just write simple text strings instead of using outsourced references.)

The *DescriptionURL* 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 now looks as follows:
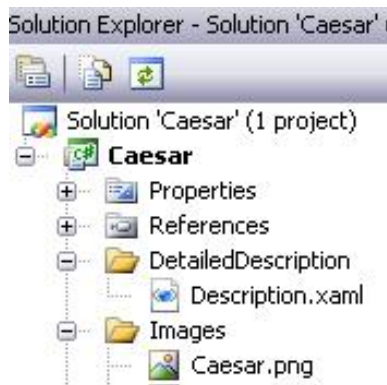
Figure 2.5.: The folder structure as seen in the Solution Explorer.

Once a detailed description has been written in the XAML file, it can be accessed in the Cryp-Tool 2 application by right-clicking on the plugin icon in the workspace and selecting *Show description* (Figure 2.6).
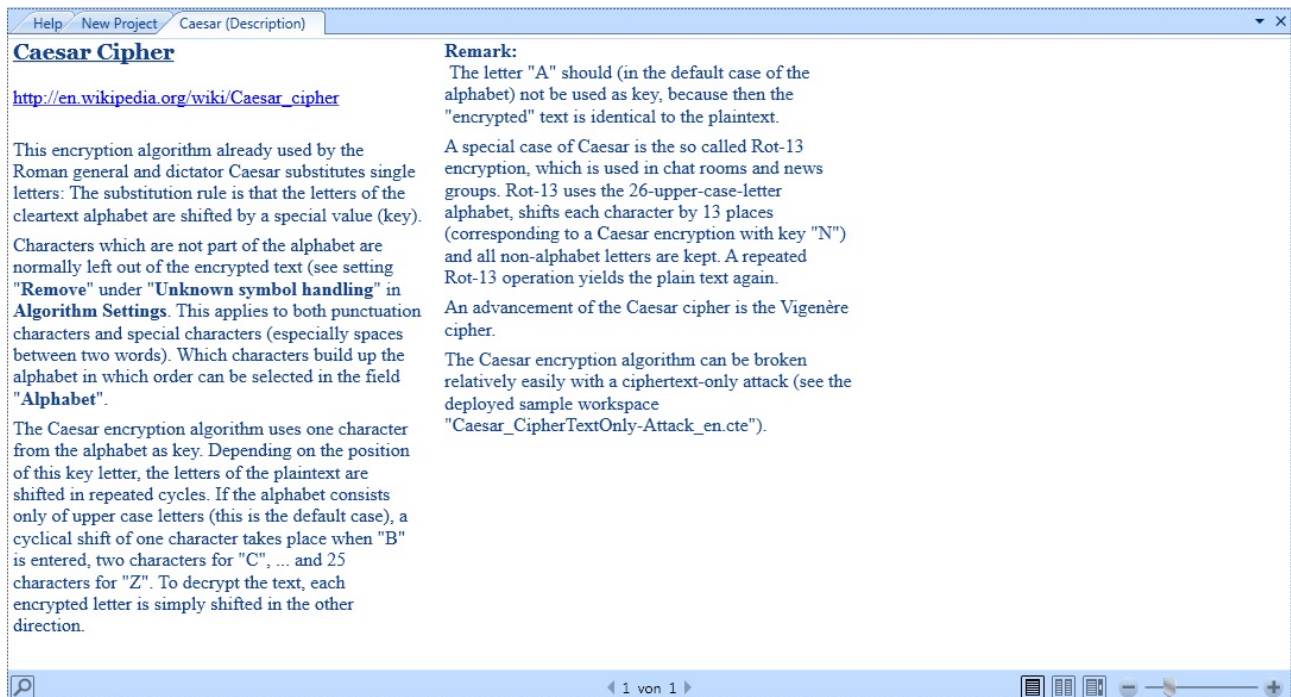
Figure 2.6.: A detailed description provided through an XAML file.

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 2.6, we must accordingly tell CrypTool 2 where to find the icon. This can be seen above in Figure 2.4.

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 choose to add more icons, don't forget to add the icons to your solution.)

2.4.3. Algorithm category and the *[EncryptionType]* attribute

In the CrypTool 2 user interface plugins are grouped by their algorithm category, for example hash, encryption, and so on. To set the category, your plugin must inherit from a specific interface, like *IHash* or *IEncryption*. Some categories require the specification of a subcategory, which is entered as an attribute[1]. In our example, Caesar inherits from *IEncryption*.

```
1    [EncryptionType(EncryptionType.Classic)]
2    public class Caesar : IEncryption
3    {
```

---

[1]The current category system is less than ideal and will be changed in future; see trac ticket #50.

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

- *Asymmetric* — for asymmetrical encryption algorithms, such as RSA.

- *SymmetricBlock* — for block cipher algorithms, such as DES, AES and Twofish.

- *SymmetricStream* — for stream cipher algorithms, such as RC4, Rabbit and SEAL.

- *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.

- *Classic* — for classical encryption or hash algorithms, such as Caesar or MD5.

2.4.4. Importing CrypPluginBase namespaces

Depending on which algorithm category you choose, you will need to import the corresponding namespace of *CrypPluginBase*. To include the necessary namespaces in the class header, use the `using` statement followed by the name of the desired namespace. *CrypPluginBase* 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, 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 standalone tools in CrypTool 2 that are not run in a workspace editor.

- *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.

2.5. 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:

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

If your algorithm deals with long 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 2.4.4.

Our example makes use of the following private variables:

- `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.

2.5.1. Adding controls to the CaesarSettings class

The settings class contains the necessary information about controls, captions, descriptions, and default parameters (for key settings, alphabets, key length, type of action, etc.) to build the settings **TaskPane** in the CrypTool application. The settings class is used to populate the TaskPane in the CrypTool 2 application so that the user can modify the plugin settings at will. Therefore, we must implement some controls, such as buttons and text boxes, to allow for the necessary interaction. If you will be implementing an algorithm that does not have any user-defined settings (such as a hash function), then this class can be left mostly empty. In Appendix A there is a full example of what a completed TaskPane and the corresponding source code for the existing Caesar plugin in CrypTool 2 looks like. You can also look at the source code of other CrypTool 2 plugins for examples of how to create the TaskPane backend.

2.6. Adding an icon to the Caesar class

Before we go back to the code of the Caesar class, we need to add a custom icon to our project to be shown in the CrypTool 2 **ribbon bar** and **navigation pane**. The template has a default icon set, so you don't need to create your own custom set.

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 with it. In the project solution, create 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*.
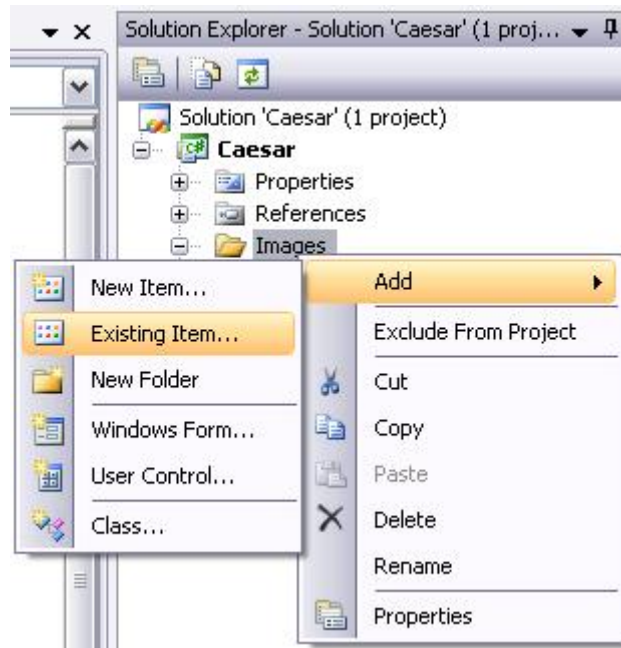
Figure 2.7.: Adding an existing item.

A new window will then appear. Select *Image Files* as the file type and select your newly-created icon for your plugin.
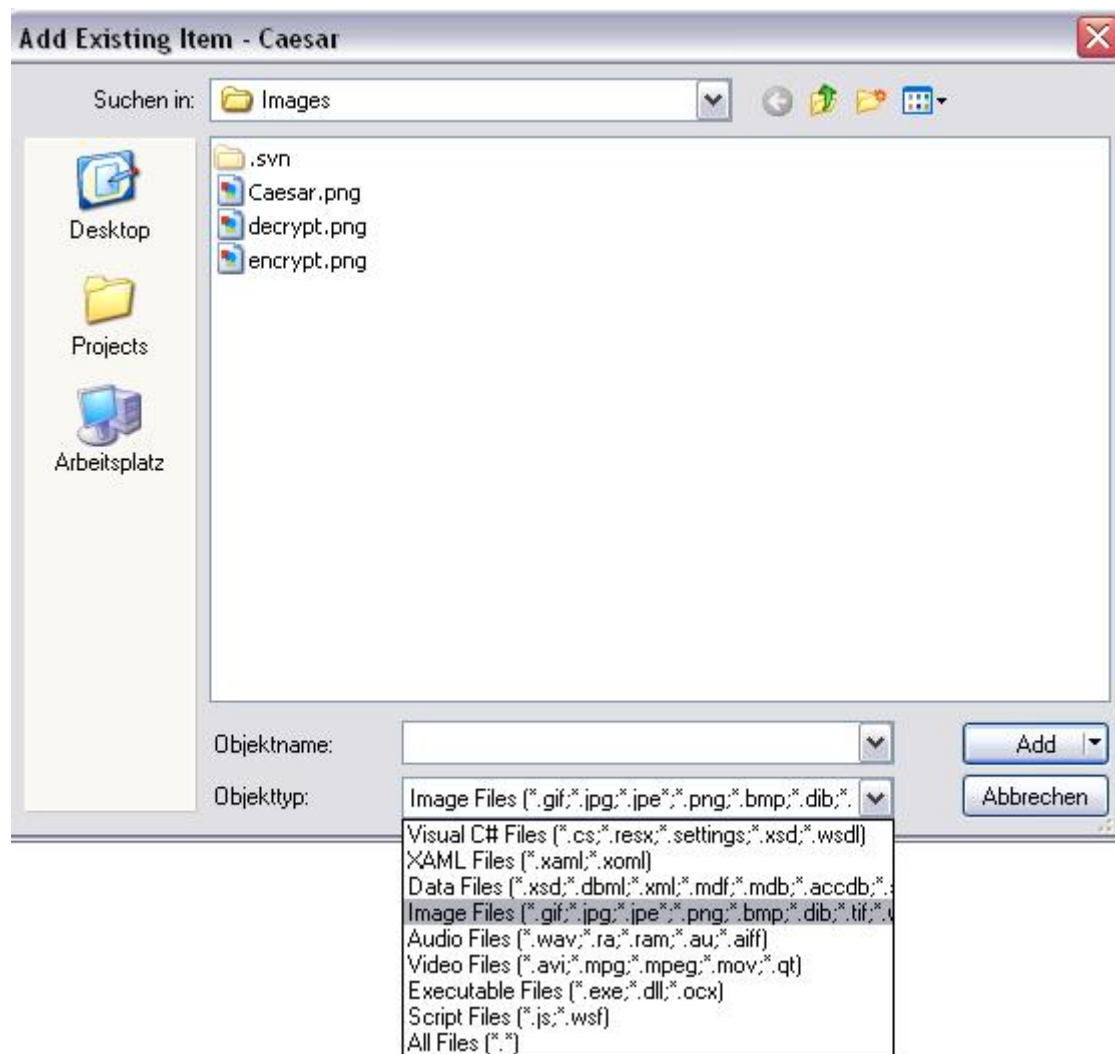
Figure 2.8.: Selecting the image file.

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.
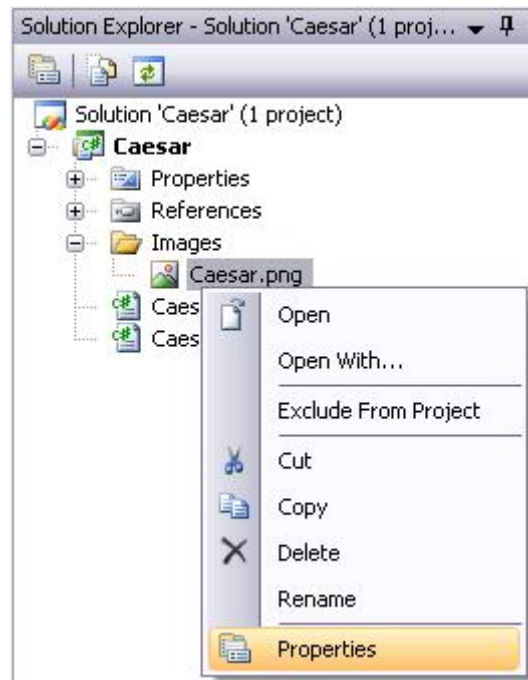
Figure 2.9.: Selecting the image properties.

In the *Properties* panel, set the *Build Action* to *Resource*.
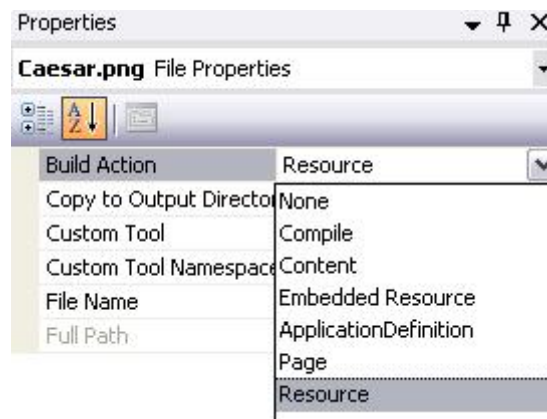
Figure 2.10.: Selecting the icon's build action.

2.7. Input and output dockpoints

2.7.1. The input/output attributes

Next we will define five properties, each with an appropriate attribute, to be used for input and output. Th attributes are necessary to tell CrypTool 2 whether the properties are used for input or 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, e.g. 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 or output arrow of the icon after it has been placed in the editor; "Input stream" in the example below:
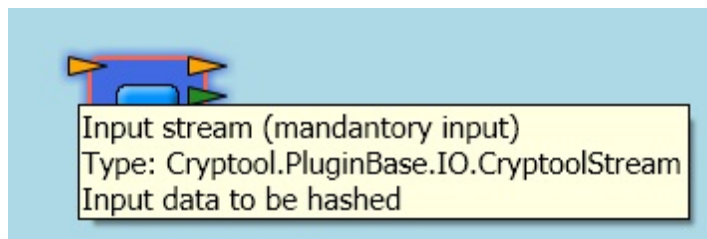
Figure 2.11.: A possible property caption and toolTip.

- *toolTip* — the toolTip for the property displayed over the input or output arrow of the icon after it has been placed in the editor; "Input data to be hashed" in the example 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 `Direction.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 using the corresponding display level will not see the properties marked as any other level, but a professional using the appropriate display level will have access to everything. These 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 quick-watch perspective. CrypTool 2 accepts the following quickwatch formats:

  - `QuickWatchFormat.Base64`
  - `QuickWatchFormat.Hex`
  - `QuickWatchFormat.None`
  - `QuickWatchFormat.Text`
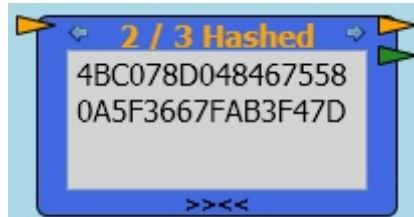
Figure 2.12.: A quickwatch display in hexadecimal.

- *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 format, such as UTF-16 or Windows-1250, you should provide here the name of a conversion method as string. The header for such a method should look something like the following:

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

2.7.2. Defining the input/output properties

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 set methods for output properties, as they will never be called from outside the plugin. Nevertheless, in our example the plugin itself accesses the property, 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, "CryptoolStream output", "The raw
    CryptoolStream data after processing with the Caesar cipher", "",
    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)]
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)", "This is
      the same setting as the shift value in the 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 }
```

2.8. Implementing the actual algorithm

Algorithmic processing should be done in the *Execute()* function. The actual functionality of your algorithm, as well as the structure thereof, is up to you. Below is our implementation of the Caesar algorithmic processing and the *Execute()* function:

```
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   // If we are working in case-insensitive mode, we will use only
8   // capital letters, hence we must transform the whole alphabet
9   // to uppercase.
10  if (!cfg.CaseSensitiveAlphabet)
11  {
12    alphabet = cfg.AlphabetSymbols.ToUpper();
13  }
14
```

```
15    if (inputString != null)
16    {
17      for (int i = 0; i < inputString.Length; i++)
18      {
19        // Get the plaintext char currently being processed.
20        char currentchar = inputString[i];
21
22        // Store whether it is upper case (otherwise lowercase is
            assumed).
23        bool uppercase = char.IsUpper(currentchar);
24
25        // Get the position of the plaintext character in the alphabet.
26        int ppos = 0;
27        if (cfg.CaseSensitiveAlphabet)
28        {
29          ppos = alphabet.IndexOf(currentchar);
30        }
31        else
32        {
33          ppos = alphabet.IndexOf(char.ToUpper(currentchar));
34        }
35
36        if (ppos >= 0)
37        {
38          // We found the plaintext character in the alphabet,
39          // hence we will commence shifting.
40          int cpos = 0;
41          switch (mode)
42          {
43            case CaesarMode.encrypt:
44              cpos = (ppos + cfg.ShiftKey) % alphabet.Length;
45              break;
46            case CaesarMode.decrypt:
47              cpos = (ppos - cfg.ShiftKey + alphabet.Length) % alphabet.
                  Length;
48              break;
49          }
50
51          // We have the position of the ciphertext character,
52          // hence just output it in the correct case.
53          if (cfg.CaseSensitiveAlphabet)
54          {
55            output.Append(alphabet[cpos]);
56          }
57          else
58          {
59            if (uppercase)
60            {
61              output.Append(char.ToUpper(alphabet[cpos]));
62            }
```

```
63            else
64            {
65                output.Append(char.ToLower(alphabet[cpos]));
66            }
67          }
68        }
69        else
70        {
71          // The plaintext character was not found in the alphabet,
72          // hence proceed with handling unknown characters.
73          switch ((CaesarSettings.UnknownSymbolHandlingMode)cfg.
                UnknownSymbolHandling)
74          {
75            case CaesarSettings.UnknownSymbolHandlingMode.Ignore:
76              output.Append(inputString[i]);
77              break;
78            case CaesarSettings.UnknownSymbolHandlingMode.Replace:
79              output.Append('?');
80              break;
81          }
82        }
83
84        // Show the progress.
85        ProgressChanged(i, inputString.Length - 1);
86      }
87      outputString = output.ToString();
88      OnPropertyChanged("OutputString");
89      OnPropertyChanged("OutputData");
90    }
91 }
92
93 public void Execute()
94 {
95    switch (settings.Action)
96    {
97      case 0:
98        GuiLogMessage("Encrypting", NotificationLevel.Debug);
99        ProcessCaesar(CaesarMode.encrypt);
100       break;
101     case 1:
102       GuiLogMessage("Decrypting", NotificationLevel.Debug);
103       ProcessCaesar(CaesarMode.decrypt);
104       break;
105     default:
106       break;
107   }
108 }
```

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* to indicate that both output properties *OutputData* and *OutputDataStream* have changed. Instead of calling the property's set method you could instead call *OnPropertyChanged* directly within the *Execute()* method.

2.8.1. Sending messages to the CrypTool 2 core

The CrypTool 2 API provides three methods to send messages from the plugin to the CrypTool 2 core. *GuiLogMessage* is used to send messages to the CrypTool 2 status bar. This method is a nice mechanism to inform the user as to what your plugin is currently doing. *OnPropertyChanged* is used to inform the core application of changes to any data output properties. This is necessary for a correct plugin execution. *ProgressChanged* is used to visualize the progress of the algorithm as a bar.
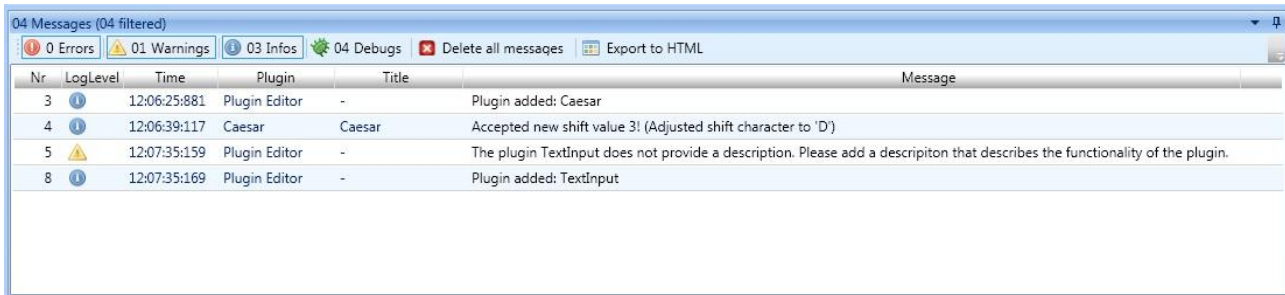
Figure 2.13.: An example status bar.

The *GuiLogMessage* method takes two parameters:

- *Message* — the text 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`

2.8.2. 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. There has been some misunderstanding about the meaning of *Dispose()*. Dispose will be called ultimately before object destruction. After disposal, the object will be in an undefined state.

```
1 public void Dispose ()
2 {
3   foreach ( CryptoolStream stream in listCryptoolStreamsOut )
4   {
5     stream . Close ();
6   }
7   listCryptoolStreamsOut . Clear ();
8 }
```

2.9. Drawing the workflow of your plugin

Each plugin should have an associated workflow file to show the algorithm in action in CrypTool 2. These workflow files are saved with the special *.cte* file extension. You can view the example files from other plugins by opening any of the files in the `ProjectSamples\` folder with CrypTool 2. Below is a sample workflow for our Caesar example:
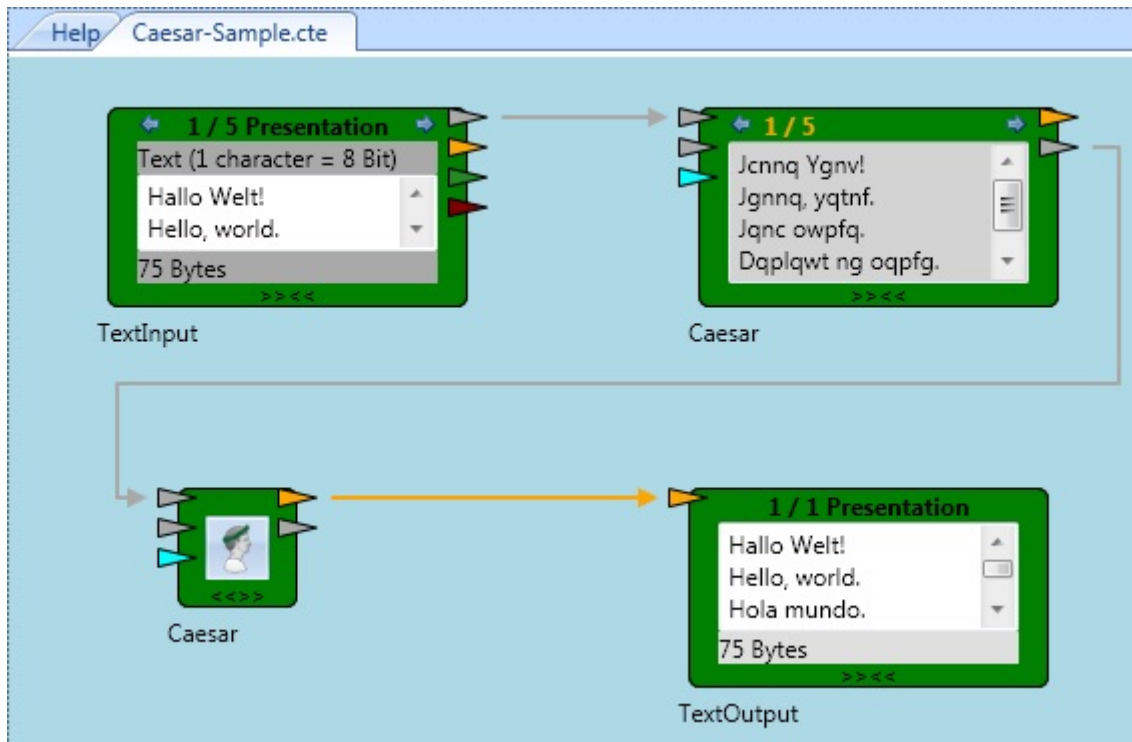
Figure 2.14.: A sample workflow diagram for the Caesar algorithm.

As your last step of development, once your plugin runs smoothly, you should also create one of these sample workflow files for your plugin. Such a file can be automatically created by simply saving a CrypTool 2 workspace project featuring your plugin. You should store the workflow file in the `ProjectSamples\` folder and make sure to commit the file to the SVN repository (see Section 1.2.3).

Appendix A.

CaesarSettings.cs
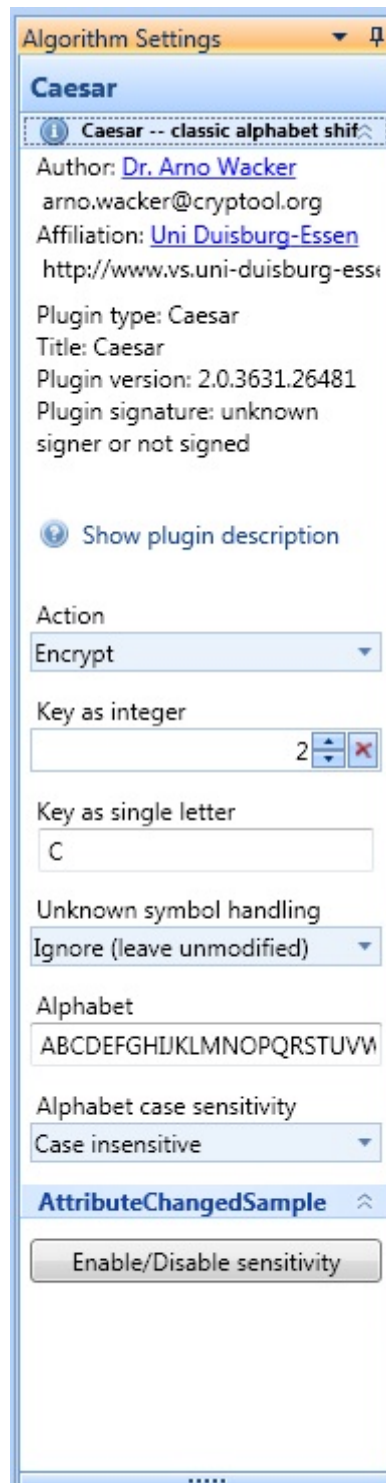
```csharp
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;

using Cryptool.PluginBase;

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

        /// <summary>
        /// This delegate is ued 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);
            }
        }

```

```csharp
48          /// <summary>
49          /// Retrieves the current setting of whether or not the
50          /// alphabet should be treated as case-sensitive.
51          /// </summary>
52          [PropertySaveOrder(1)]
53          public bool CaseSensitiveAlphabet
54          {
55              get
56              {
57                  if (caseSensitiveAlphabet == 0)
58                  {    return false;    }
59                  else
60                  {    return true;      }
61              }
62              set {} // this setting is readonly, but we must include
63                     // some form of set method to prevent problems.
64          }
65
66          /// <summary>
67          /// Returns true if any settings have been changed.
68          /// This value should be set externally to false, i.e.
69          /// when a project is saved.
70          /// </summary>
71          [PropertySaveOrder(3)]
72          public bool HasChanges
73          {
74              get { return hasChanges; }
75              set { hasChanges = value; }
76          }
77
78          #endregion
79
80          #region Private variables
81          private bool hasChanges;
82          private int selectedAction = 0;
83          private string upperAlphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
84          private string lowerAlphabet = "abcdefghijklmnopqrstuvwxyz";
85          private string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
86          private char shiftChar = 'C';
87          private int shiftValue = 2;
88          private UnknownSymbolHandlingMode unknownSymbolHandling =
                  UnknownSymbolHandlingMode.Ignore;
89          private int caseSensitiveAlphabet = 0; // 0 = case-insensitve,
                  1 = case-sensitive
90          private bool sensitivityEnabled = true;
91          #endregion
92
93          #region Private methods
94
95          private string removeEqualChars(string value)
```

```csharp
 96            {
 97                int length = value.Length;
 98
 99                for (int i = 0; i < length; i++)
100                {
101                    for (int j = i + 1; j < length; j++)
102                    {
103                        if ((value[i] == value[j]) || (!
                               CaseSensitiveAlphabet & (char.ToUpper(value[i])
                                == char.ToUpper(value[j]))))
104                        {
105                            LogMessage("Removing duplicate letter: \'" +
                                   value[j] + "\' from alphabet!",
                                   NotificationLevel.Warning);
106                            value = value.Remove(j,1);
107                            j--;
108                            length--;
109                        }
110                    }
111                }
112
113                return value;
114            }
115
116            /// <summary>
117            /// Set the new shiftValue and the new shiftCharacter
118            /// to offset % alphabet.Length.
119            /// </summary>
120            private void setKeyByValue(int offset)
121            {
122                HasChanges = true;
123
124                // Make sure the shift value lies within the alphabet
                     range.
125                offset = offset % alphabet.Length;
126
127                // Set the new shiftChar.
128                shiftChar = alphabet[offset];
129
130                // Set the new shiftValue.
131                shiftValue = offset;
132
133                // Announce this to the settings pane.
134                OnPropertyChanged("ShiftValue");
135                OnPropertyChanged("ShiftChar");
136
137                // Print some info in the log.
138                LogMessage("Accepted new shift value " + offset + "! (
                     Adjusted shift character to \'" + shiftChar + "\')",
                     NotificationLevel.Info);
```

```csharp
139             }
140
141         private void setKeyByCharacter ( string value )
142         {
143             try
144             {
145                 int offset ;
146                 if ( this . CaseSensitiveAlphabet )
147                 {
148                     offset = alphabet . IndexOf ( value [0]) ;
149                 }
150                 else
151                 {
152                     offset = alphabet . ToUpper () . IndexOf ( char . ToUpper (
                         value [0]) ) ;
153                 }
154
155                 if ( offset >= 0)
156                 {
157                     HasChanges = true ;
158                     shiftValue = offset ;
159                     shiftChar = alphabet [ shiftValue ];
160                     LogMessage ( "Accepted new shift character \'" +
                         shiftChar + "\'! ( Adjusted shift value to " +
                         shiftValue + ")" , NotificationLevel . Info ) ;
161                     OnPropertyChanged ( "ShiftValue" ) ;
162                     OnPropertyChanged ( "ShiftChar" ) ;
163                 }
164                 else
165                 {
166                     LogMessage ( "Bad input \"" + value + "\"! (
                         Character not in alphabet !) Reverting to " +
                         shiftChar . ToString () + "!" , NotificationLevel .
                         Error ) ;
167                 }
168             }
169             catch ( Exception e )
170             {
171                 LogMessage ( "Bad input \"" + value + "\"! (" + e .
                     Message + ") Reverting to " + shiftChar . ToString ()
                     + "!" , NotificationLevel . Error ) ;
172             }
173         }
174
175     #endregion
176
177     #region Algorithm settings properties ( visible in the Settings
            pane )
178
179     [ PropertySaveOrder (4) ]
```

```
180        [ContextMenu("Action", "Select the algorithm action", 1,
              DisplayLevel.Beginner, ContextMenuControlType.ComboBox, new
               int[] { 1, 2 }, "Encrypt", "Decrypt")]
181        [TaskPane("Action", "setAlgorithmActionDescription", null, 1,
              true, DisplayLevel.Beginner, ControlType.ComboBox, new
              string[] { "Encrypt", "Decrypt" })]
182        public int Action
183        {
184            get
185            {
186                return this.selectedAction;
187            }
188            set
189            {
190                if(value != selectedAction)
191                {
192                    HasChanges = true;
193                    this.selectedAction = value;
194                    OnPropertyChanged("Action");
195                }
196                if(ReExecute != null)
197                { ReExecute(); }
198            }
199        }
200
201        [PropertySaveOrder(5)]
202        [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)]
203        public int ShiftValue
204        {
205            get { return shiftValue; }
206            set
207            {
208                setKeyByValue(value);
209                if (ReExecute != null)
210                { ReExecute(); }
211            }
212        }
213
214        [PropertySaveOrder(6)]
215        [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}")]
216     public string ShiftChar
217     {
218         get { return this.shiftChar.ToString(); }
219         set
220         {
221             setKeyByCharacter(value);
222             if (ReExecute != null)
223                 {   ReExecute();    }
224         }
225     }
226
227     [PropertySaveOrder(7)]
228     [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 \'?\'"
            })]
229     [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 \'?\'" })]
230     public int UnknownSymbolHandling
231     {
232         get { return (int)this.unknownSymbolHandling; }
233         set
234         {
235             if((UnknownSymbolHandlingMode)value !=
                    unknownSymbolHandling)
236             {
237                 HasChanges = true;
238                 this.unknownSymbolHandling = (
                        UnknownSymbolHandlingMode)value;
239                 OnPropertyChanged("UnknownSymbolHandling");
240         }
241             if (ReExecute != null)
242             { ReExecute();   }
243         }
244     }
245
246     [SettingsFormat(0, "Normal", "Normal", "Black", "White",
            Orientation.Vertical)]
247     [PropertySaveOrder(9)]
248     [TaskPane("Alphabet", "This is the alphabet currently in use."
            , null, 6, true, DisplayLevel.Expert, ControlType.TextBox,
            "")]
249     public string AlphabetSymbols
250     {
```

```
251            get { return this.alphabet; }
252            set
253            {
254              string a = removeEqualChars(value);
255              if (a.Length == 0) // cannot accept empty alphabets
256              {
257                LogMessage("Ignoring empty alphabet from user! Using
                      previous alphabet instead: \" + alphabet + "\" (" +
                      alphabet.Length.ToString() + " Symbols)",
                      NotificationLevel.Info);
258              }
259              else if (!alphabet.Equals(a))
260              {
261                HasChanges = true;
262                this.alphabet = a;
263                setKeyByValue(shiftValue); // reevaluate if the
                      shiftvalue is still within the range
264                LogMessage("Accepted new alphabet from user: \"" +
                      alphabet + "\" (" + alphabet.Length.ToString() + "
                      Symbols)", NotificationLevel.Info);
265                OnPropertyChanged("AlphabetSymbols");
266
267                if (ReExecute != null)
268                { ReExecute();  }
269              }
270            }
271          }
272
273          /// <summary>
274          /// Visible setting how to deal with alphabet case.
275          /// 0 = case-insentive , 1 = case-sensitive
276          /// </summary>
277          [PropertySaveOrder(8)]
278          [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" })]
279          [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" })]
280          public int AlphabetCase
281          {
282              get { return this.caseSensitiveAlphabet; }
283              set
284              {
285                  if (value != caseSensitiveAlphabet)
286                  { HasChanges = true;  }
287                  this.caseSensitiveAlphabet = value;
288                  if (value == 0)
```

```csharp
289                     {
290                         if (alphabet == (upperAlphabet + lowerAlphabet))
291                         {
292                             alphabet = upperAlphabet;
293                             LogMessage("Changing alphabet to: \"" +
                                     alphabet + "\" (" + alphabet.Length.
                                     ToString() + " Symbols)", NotificationLevel
                                     .Info);
294                             OnPropertyChanged("AlphabetSymbols");
295                             // reset the key (shiftvalue/shiftChar)
296                             // to be in the range of the new alphabet.
297                             setKeyByValue(shiftValue);
298                         }
299                     }
300                     else
301                     {
302                         if (alphabet == upperAlphabet)
303                         {
304                             alphabet = upperAlphabet + lowerAlphabet;
305                             LogMessage("Changing alphabet to: \"" +
                                     alphabet + "\" (" + alphabet.Length.
                                     ToString() + " Symbols)", NotificationLevel
                                     .Info);
306                             OnPropertyChanged("AlphabetSymbols");
307                         }
308                     }
309
310                     // Remove equal characters from the current alphabet.
311                     string a = alphabet;
312                     alphabet = removeEqualChars(alphabet);
313                     if (a != alphabet)
314                     {
315                         OnPropertyChanged("AlphabetSymbols");
316                         LogMessage("Changing alphabet to: \"" + alphabet +
                                 "\" (" + alphabet.Length.ToString() + "
                                 Symbols)", NotificationLevel.Info);
317                     }
318                     OnPropertyChanged("AlphabetCase");
319                     if (ReExecute != null)
320                     { ReExecute();  }
321                 }
322             }
323
324         #endregion
325
326         #region INotifyPropertyChanged Members
327
328         public event PropertyChangedEventHandler PropertyChanged;
329
330         protected void OnPropertyChanged(string name)
```

```csharp
331         {
332           if (PropertyChanged != null)
333           {
334             PropertyChanged(this, new PropertyChangedEventArgs(name));
335           }
336         }
337
338         #endregion
339
340         #region TaskPaneAttributeChanged (Sample)
341         /// <summary>
342         /// This event is here merely as a sample.
343         /// </summary>
344         public event TaskPaneAttributeChangedHandler
                 TaskPaneAttributeChanged;
345
346         [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)]
347         public void EnableDisableSesitivity()
348         {
349           if (TaskPaneAttributeChanged!= null)
350           {
351             sensitivityEnabled = !sensitivityEnabled;
352             if (sensitivityEnabled)
353             {
354               TaskPaneAttributeChanged(this, new
                     TaskPaneAttributeChangedEventArgs(new
                     TaskPaneAttribteContainer("AlphabetCase", Visibility.
                     Visible)));
355             }
356             else
357             {
358               TaskPaneAttributeChanged(this, new
                     TaskPaneAttributeChangedEventArgs(new
                     TaskPaneAttribteContainer("AlphabetCase", Visibility.
                     Collapsed)));
359             }
360           }
361         }
362         #endregion TaskPaneAttributeChanged (Sample)
363     }
364 }
```