The SharpDevelop add-in tree architecture.

Version 0.9.a

Mike Krueger
Bernd Spuida (ed)

February 26, 2002

Contents

1 The problem of add-in architectures

Almost all mid to large size software projects have some sort of **add-in** architecture. An **add-in** is basically an extension to the functionality of the main application.
The common way to introduce an **add-in** structure is to load libraries from a specific directory at the runtime. These libraries usually add actions to a menu and the user then may select these actions. Such an action performs something on the screen or opens a given file format or some other specific task. In general, these actions do only one or two specific jobs they are limited to.
This approach is easy to implement after an application has been written and needs to be extended with additional **add-ins**. But it is not very flexible, because using this approach:

- **add-ins** cannot extend other parts outside "add-in space"
- **add-ins** cannot extend each other
- **add-ins** are external programs and do not extend **Core** functionality

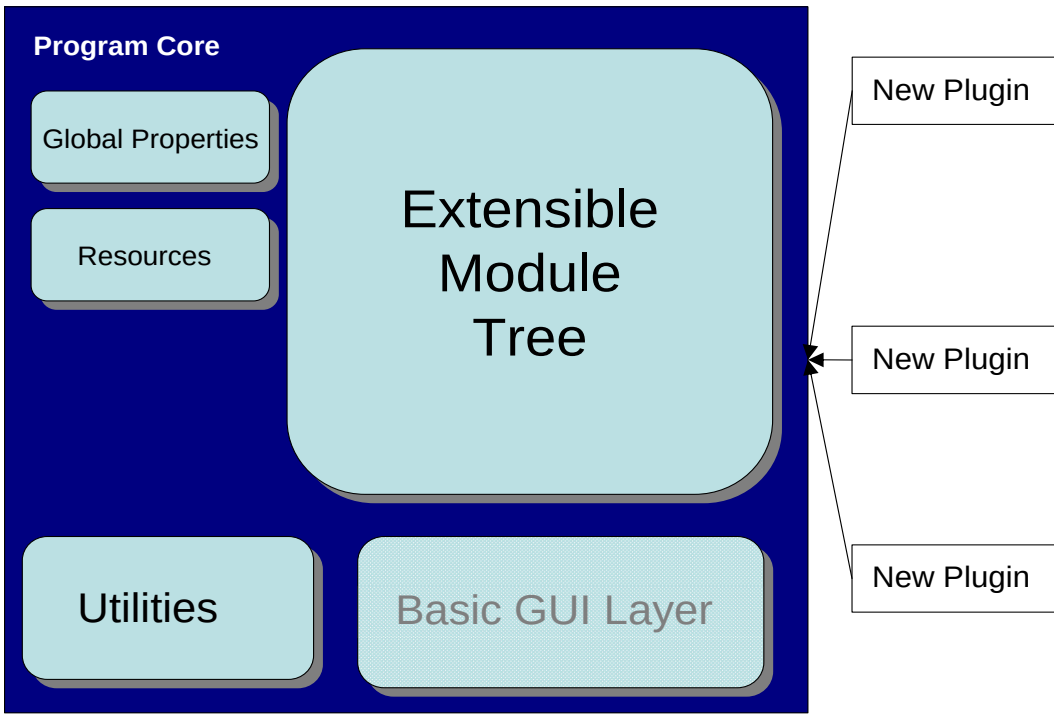
1.1 The idea

The approach I decided to take attempts to make **add-ins** something more versatile than the external programs they commonly are. I wanted a small **Core** code and all additional functionality ought to then be defined by external **add-ins** co-operating with each other and extending their mutual capabilities. The approach should be simple, extensible and not dependent on the type of application it is used for[1].
I realized that the only way of doing a clean **add-in** system is to design the application from the ground up to depend on an **add-in** model.

1.2 Platform structure

The application itself is based on a small **Core** System. All functionality is implemented in one or more **add-ins** which are plugged into the **Core**.
Inside the **Core** System some basic subsystems ought to be defined which extend the **Core** functionality.

[1] But generally it ought to be an application with windows displaying content etc. and not a screen-wide app such as a computer game.



The Subsystems[2] and **add-ins** define an extensible tree structure. The **paths**[3] which may be extended are well defined by the application which is built on the top of the core system. At these **path** nodes, **add-ins** can contribute functionality.

Each major component of the application implements some key function and defines new tree nodes.

1.2.1 Program Core

The **core** of the platform implements the **add-in** loader which loads **add-ins** dynamically. A **add-in** is a component which is defined in a manifest xml file.
The **core** keeps track of the installed **add-ins** and the functions they provide. Functions are added to the **core** using a common directory model. The paths lead to well defined extension points where **add-ins** contribute functionality to or add extensions to the platform. By adding extensions to the tree, **add-ins** may allow to be extended in their turn by other **add-ins**.
Extending the tree is the only mechanism for adding functionality to the platform or to other **add-ins**. All **add-ins** use the same mechanism.
Written specifically for the .NET Platform, the **core** can not only be extended in the C♯

[2] Components of the application residing outside the **Program Core.**
[3] A path representation for finding nodes in the tree is used. Similar to UNIX paths, a / character is used as the separator. A path represents a traverals of the tree from its root to a give destination node.

language, **add-ins** may be written in any other .NET language like VB, Java or C++ as well, including languages implemented by third parties.

1.2.2 Global Properties

The **properties** engine implements a way of storing all forms of configuration variables. There is an application wide property class which stores custom **properties** in a file in the personal folder of the current user. Properties will be discussed in detail in chapter 4.

1.2.3 Resources

Modern applications need the capability of using localized resources. The **Core** defines a class which allows localized strings and images for the whole application. The details of this localization class are given in chapter 5.1.

1.2.4 GUI Layer

It is always a good idea to abstract the given GUI API somewhat towards the needs of the application. Therefore the **Core** does have a thin layer wrapping the Windows.Forms API. This facilitates possible conversion of the Application onto other GUI toolkits. Documentation of this layer still is heavily under construction. The GUI Layer will be discussed in a separate document.

1.2.5 Utilities

For an overview of the utilities integrated in the program **Core**, see chapter 5.

2 Definition of the add-in architecture using Xml

2.1 Introduction

Xml is a good way to describe hierarchies and xml definitions are generally extensible. This is an important factor for customizing the xml for other projects or for extensions to the first xml format.

One important goal of the xml definition is that all **add-ins** and **Core** components should share the same format. But in contrast to that goal, **add-ins** should be able to extend the format according to their needs; for **add-ins might need extended interfaces in their xml. Extensions to the interface must be described in the documentation for the add-in in question.**

2.2 Example xml

We will start with a small example xml file. I started the design of the structure too by defining a similar small sample xml.

```
<AddIn name        = "NCvs Core"
       author      = "Author"
       copyright   = "GPL"
       url         = "http://www.icsharpcode.net"
       description = "NCvs core codon"
       version     = "1.0.0">

  <Requires>
    <codon path="/NCvs/Workspace"/>
  </Requires>

  <Runtime>
    <Import assembly="NCvs.RepositoryView.dll"/>
  </Runtime>

  <Extension path = "/NCvs/Workspace/MainMenu">
    <MenuItem id = "File" label = "&amp;File">
      <MenuItem id = "Separator1" label = "-"/>
      <MenuItem id    = "Exit"
                label = "E&amp;xit"
                class = "NCvs.Gui.Commands.ExitCommand"/>
    </MenuItem>
  </Extension>
</AddIn>
```

Now examining the xml given, we notice different sections of the file representing a number of concepts.

The first section is the root xml node. It has several attributes which can be used in an add-in manager (for installation, removal, or updating of add-ins).

The `Requires` xml node serves a special purpose, telling the add-in loader which paths must exist in the tree to ensure that this codon[1] is loaded correctly. The tree can be traversed using paths similar to unix paths. It has a root at / and underlying "directories" form the branches of the tree. In

[1] A **codon** is any element inserted into the tree, not necessarily a program extension by means of program code. It may as well be something such as a file import filter, a graphics resource etc. This abstract term was choosen to avoid using terminology such as 'module', which certainly has too many connotations depending on the personal experience of the reader.

contrast to a file system path, the "files" (called codons) may have "sub-directories". A tree node does not necessarily contain a codon, it may be empty and thus just be a "directory".

The next xml node is `Runtime`. It exists because the executable code needs to reside in some place. The xml definition shouldn't actually contain an executable program, it defines where the binary is located. As many assemblies as wanted may be loaded by specifing more than one `Import` node.

The next xml node is very special. It defines an entry point in the tree and its sub items are the actual codons used in the program. These codons are program specific and make up the heart of the xml definition.

2.3 Codons overview

While examining the codons, we have seen that they have attributes specifying the behaviour or state of some element of the program.

There are some common attributes for each codon (such as the `id` attribute). But optional attributes exist as well (such as the `class` attribute). This results in several general attribute classes: *common attributes*, *required common attributes*, *optional attributes* **and** *required optional attributes*. **All these attribute classes should be supported.**

Every codon needs an ID to be identified in the tree; the codon is then inserted into in the tree beneath the `extension path + '/' + id`. Therefore it is a necessity for all codons and is a required common attribute.
The `label` attribute of the MenuItem is a required attribute for the `MenuItem` codon. The `class` attribute might be considered an optional attribute of the `MenuItem` codon, but as the case that a codon specifies a class as an option is very common, it is a common attribute.

The `MenuItem` codon is a special case of the more general concept of what a codon is. Let us consider an object of which we have more than one instance and which we can put in a collection, wanting that someone else can extend it[2]. It is desirable to have these objects in the codon form, therefore the codon definition should be extensible by add-ins without digging deep into the xml definition.

Note: It is possible to define almost anything in a codon. We can define not only GUI elements and their actions, it is possible to extend algorithms

[2] If you find no example, take one of these: toolbar buttons, compression algorithms, file filters, dialog panels in wizards or option dialogs etc.

or behaviours of complex subsystems as well.

2.3.1 Arranging codons

One requirement for menu items is that they need to be in some sort of order. It certainly makes sense when our menu item from codon Y is inserted between menu item x from codon Z and menu item y from the Core codon. When we have only a single xml file defining the menu this is no problem. Problems arise when the definitions are scattered across more than one xml file[3].
Therefore all codons have the common attributes `insertafter` and `insertbefore` with `<MenuItem id = "MyMenu" insertafter = "File" label = "MyMenu">`. This ensures that our menu item is inserted after the File menu.

The `insertbefore` attribute works like the `insertafter` attribute. More than one codon may be specified by using a comma as name separator. You must specify the ID of the codon which you want your codon to be inserted after. The ID is the identifing attribute of a node. We have to ensure that two nodes in the same tree level never have identical IDs,as as this will make the application throw an exception. Though the codons will be topologically sorted, keep in mind that there may be more than one valid topological sort of the codons. Sometimes you may see unpredictable results inserting a codon, as it might be inserted at some distance from the codon after which it should be inserted. 'After' does not imply an actual insertion point immediately after the first (theoretically) possible position (the same applies to the before tag).

2.3.2 More than one level of codons

In the case of menu items we have more than one level of the Subtree MainMenu containing the menu definition. In the above example, I used a more concise way for describing this :

```
<Extension path = "/NCvs/Workspace/MainMenu">
  <MenuItem id = "File" label = "&amp;File"/>
</Extension>
<Extension path = "/NCvs/Workspace/MainMenu/File">
  <MenuItem id = "Separator1" label = "-"/>
  <MenuItem id    = "Exit"
            label = "E&amp;xit"
            class = "NCvs.Gui.Commands.ExitCommand"/>
  </MenuItem>
```

[3]This is the case in real life, as **add-ins** are written by a number of individuals and distributed separately, each with its own xml manifest file.

```
</Extension>
```

using instead:

```
<Extension path = "/NCvs/Workspace/MainMenu">
  <MenuItem id = "File" label = "&amp;amp;File">
    <MenuItem id = "Separator1" label = "-"/>
    <MenuItem id    = "Exit"
              label = "E&amp;amp;xit"
              class = "NCvs.Gui.Commands.ExitCommand"/>
  </MenuItem>
</Extension>
```

We see that the File MenuItem codon does not contain the submenu item codons, these are instead stored in the tree as children of the "File" codon ID. Using the short form is not mandatory, but it will make the definitions much more readable.

2.4 Conditions

It is desirable that some menu items are only displayed when some state is active, or when a specific add-in is installed. On the other hand, we might want to disable menu items when something is inactive. For these cases, conditions are introduced to the add-in structure. Conditions are a more general concept, though. They are not only useful for menu items they can be used in other situations as well.

2.4.1 Example

We might want to extend the MainMenu of our application with a Window Menu containing options for tiling horizontally, cascading and so on, but we have different versions of the workspace window like MDI, floating windows or SDI window to deal with.

When we have the SDI workspace window active, we would not want the window menu to be displayed.

This might be done as in the following example :

```
<Conditional string="${property:NCvs.Gui.Workspace}" equals="MDI">
  <MenuItem id = "Window" label = "&amp;amp;Window">
    <MenuItem id = "TileH"   label = "Tile horizontal"/>
    <MenuItem id = "TileV"   label = "Tile vertical"/>
    <MenuItem id = "Cascade" label = "Cascade"/>
  </MenuItem>
</Conditional>
```

In this case, the whole menu is stripped out of the menu bar. But in another use case, when the MDI window is active but no child window is open in the main window, the actions in the window menu may not be performed. Most programs do not strip out actions, they just disable the corresponding menu items.

In such a case, something like the following example might be done:

```
<Conditional string="${property:NCvs.Gui.Workspace}" equals="MDI">
  <MenuItem id = "Window" label = "&amp;Window">
    <Conditional activemdi="*" action="Disable">
      <MenuItem id = "TileH"   label = "Tile horizontally"/>
      <MenuItem id = "TileV"   label = "Tile vertically"/>
      <MenuItem id = "Cascade" label = "Cascade"/>
    </Conditional>
  </MenuItem>
</Conditional>
```

This may not be the last special condition we might need, therefore conditions need to be as extensible as codons.

2.4.2 Required attributes

As you can see, the `Conditional` nodes do not have different names the way codons have.
How can conditions then be distinguished from each other?
The answer is: they have all different required attributes. They may have the same required attributes but the set of all required attributes for each condition must be unique.

Note: No two conditions with the same attributes may be defined.

2.4.3 Complex conditions

Conditions may be combined with each other. We can combine them logically with *and* using child conditions, but what to do when we want one condition to match another using *or*? One possible approach would be to write two condition nodes and simulate an `if ...else` construct, but the problem with this approach becomes apparent when we want to negate a condition, as we have no tool for condition negation so far. To make condition handling a bit easier, a solution exists for combining multiple conditions in a single term.

We define a term as:

$$< Condition/ >$$

$$< Or > term_1, term_2, \cdots, term_n < /Or >$$

$$< And > term_1, term_2, \cdots, term_n < /And >$$

$$< Not > term < /Not >$$

This is the minimum set of operators needed to combine conditions. Possibly more logical operators will be added later.

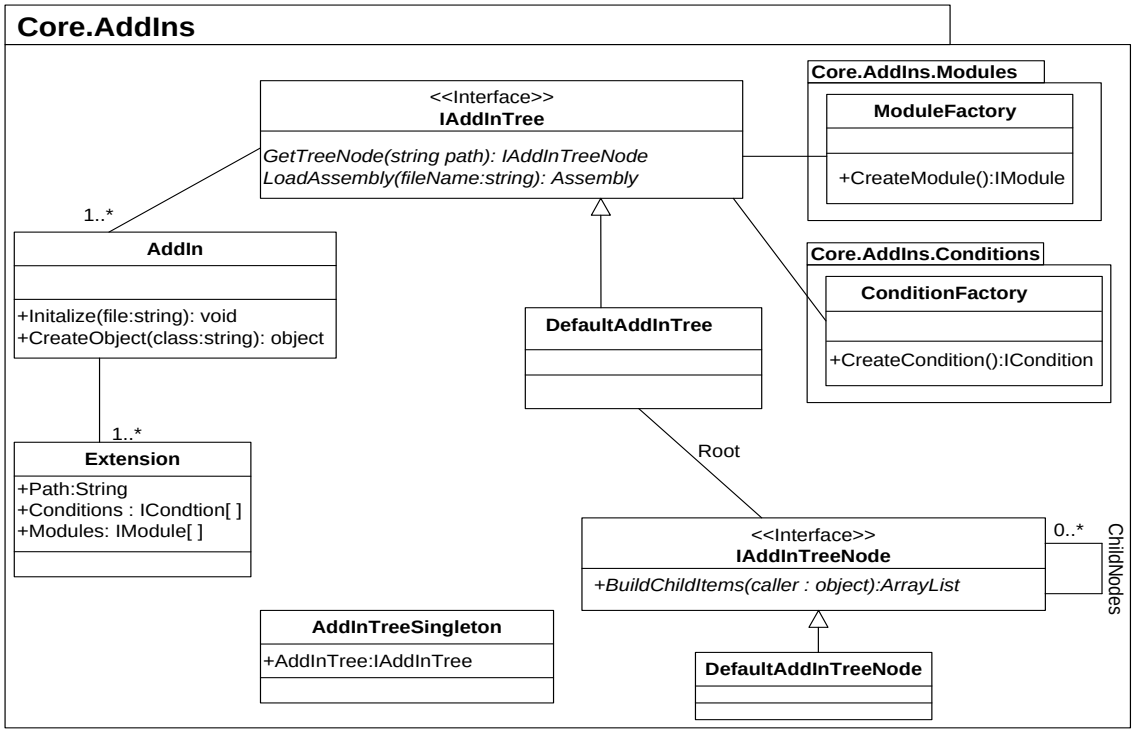
In Xml, complex conditional terms look like this:

```
<Conditional>
  <Or>
    <Condition ownerstate="RepositorySelected"/>
    <Condition ownerstate="FolderSelected"/>
  </Or>

  <Conditional ownerstate="RepositorySelected" action="Disable">
    <MenuItem id = "Update"
              label = "&amp;Update"
              class = "NCvsRepositoryView.Commands.UpdateCommand"/>
    <MenuItem id = "Commit"
              label = "&amp;Commit"
              class = "NCvsRepositoryView.Commands.CommitCommand"/>
  </Conditional>
  <MenuItem id = "Separator1" label = "-"/>
  <MenuItem id = "Import"
            label = "&amp;Import"
            class = "NCvsLocalView.Commands.ImportCommand"/>
</Conditional>
```

This menu is only active when the owner is either in the state `RepositorySelected` or in the state `FolderSelected` and the items `Update` and `Commit` will only be active when the state is `RepositorySelected`. The action attribute of the combined condition may be defined as usual in the `Conditional` node.
The operators $< And/ >$, $< Or/ >$ and $< Not/ >$ are defined. The operators are not extensible by add-ins, the Core defines the set of valid operators.

Note: The complex conditional node may contain only one term.

3 Platform architecture



The most important class is the `AddInTree`. It manages all add-ins, assembly[1] loading, codon and condition creation.

[1] An assembly is the binary format of the .NET runtime executables. It contains compiled classes.

3.1 Overview of the Core.AddIns namespace

3.1.1 Definition of the IAddInTree interface

The tree uses the following interface:

```
public interface IAddInTree
{
  // Returns the default condition factory. ICondition objects
  // are created only with this factory during the tree
  // construction process.
  ConditionFactory ConditionFactory {
    get;
  }

  // Returns the default codon factory. ICodon objects
  // are created only with this factory during the tree
  // construction process.
  CodonFactory CodonFactory {
    get;
  }

  // Returns a collection of all loaded add-ins.
  AddInCollection AddIns {
    get;
  }

  // Returns an IAddInTreeNode corresponding to path.
  IAddInTreeNode GetTreeNode(string path);

  // Inserts an AddIn into the AddInTree.
  void InsertAddIn(AddIn addIn);

  // Removes an AddIn from the AddInTree.
  void RemoveAddIn(AddIn addIn);

  // This method loads an assembly and
  // gets all its defined codons and conditions
  Assembly LoadAssembly(string name);
}
```

This is the interface the DefaultAddInTree exposes to the program.

The AddIns collection contains all `AddIn` objects which are currently loaded. The `AddInTreeSingleton` loads all add-ins from a default directory and its subdirectories. Dynamic loading is also possible using the `InsertAddIn` method. Dynamic unloading can be done by the `RemoveAddIn` method. An exception is thrown when trying to remove an add-in on which another currently installed add-in depends. All extension points which are defined by this add-in are removed. It is maybe necessary to raise an event for each removed extension

point. This feature may be added later, if this should prove to be the case. The removal of currently open workbench windows which are opened by the add-in removed is not implemented currently and is to be worked on. Removal of the add-in menu entries, buttons etc. should be done by a GUI rebuild event.

3.1.2 Definition of the IAddInTreeNode interface

```
// This interface represents an interface to tree nodes in the
// IAddInTree
public interface IAddInTreeNode
{
  // Returns a hash table containing the child nodes. Where the key
  // is the node name and the value is an AddInTreeNode object.
  Hashtable ChildNodes {
    get;
  }

  // Returns a codon defined in this node, or null if no codon
  // was defined.
  ICodon Codon {
    get;
  }

  // Returns all conditions for this TreeNode.
  ConditionCollection ConditionCollection {
    get;
  }

  // Builds all child items of this node using the BuildItem
  // method of each codon in the child tree.
  ArrayList BuildChildItems(object owner);
}
```

The `IAddInTreeNode` is the interface between the AddInTree and the codons. With a call to the `BuildChildItems` method we obtain an ArrayList of built codon items. Each codon does build an item. An item may be the GUI menu item or the codon itself. Using the build method of the codon the codon gets it current child nodes and it is up to the codon to decide what it actually does build. Refer to the respective codon documentation[2] for more information about the items which are built.

[2]The **codon** documentation of course is the responsibility of the **codon**'s author.

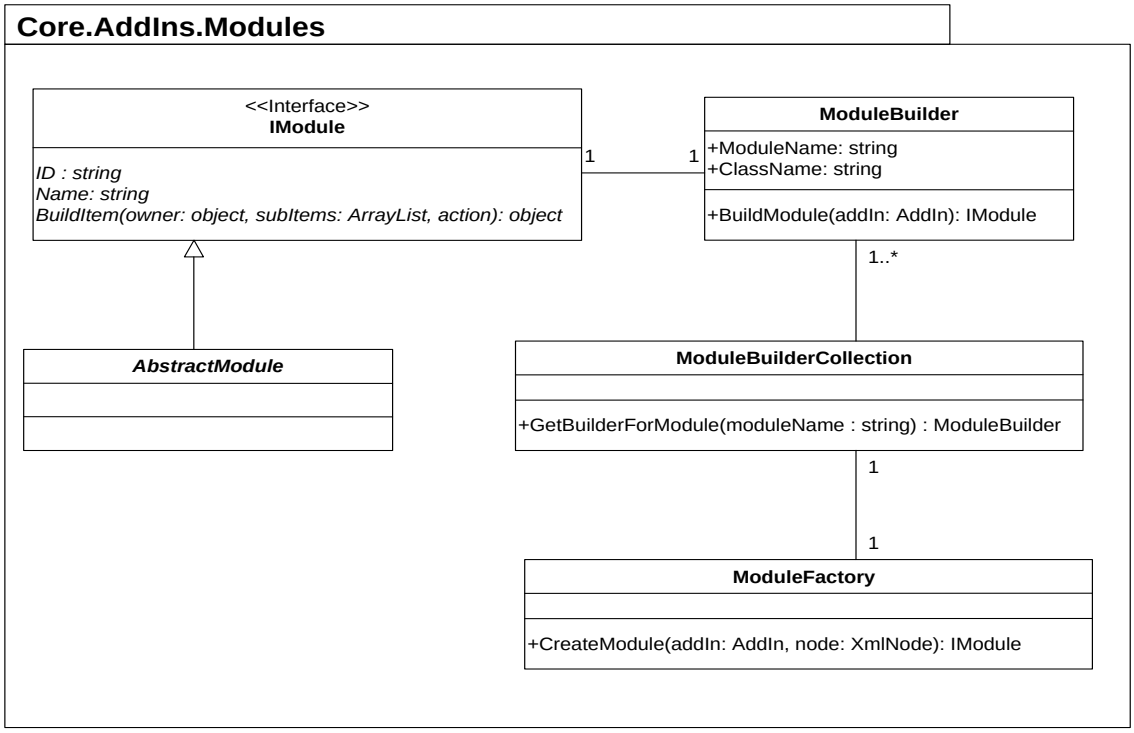
3.1.3 Definition of the AddIn and Extensions class

```
public class AddIn
{
  // returns the Name of the AddIn
  public string Name {
    get;
  }

  // returns the Author of the AddIn
  public string Author {
    get;
  }

  // returns a copyright string of the AddIn
  public string Copyright {
    get;
  }

  // returns an url of the homepage of the add-in
  // or the author.
  public string Url {
    get;
  }

  // returns a brief description of what the add-in
  // does.
  public string Description {
    get;
  }

  // returns the version of the add-in.
  public string Version {
    get;
  }

  // returns the required codons
  public StringCollection RequiredCodons  {
    get;
  }

  // returns a hashtable with the runtime libraries
  // where the key is the assembly name and the value
  // is the assembly object.
  public Hashtable RuntimeLibraries {
    get;
  }

  // returns an arraylist with all extensions defined by
```

```
  // this addin.
  public ArrayList Extensions {
    get;
  }

  // Initializes this addIn. It loads the xml definition in file
  // fileName.
  public void Initialize(string fileName);

  // Creates an object which is related to this Add-In.
  public object CreateObject(string className);

  // Defines an extension point (path in the tree) with
  // its codons.
  public class Extension
  {
     // returns the path in which the underlying codons
     // are inserted
     public string Path {
       get;
       set;
     }

     // returns a Hashtable with all conditions defined in
     // this extension.
     // where the key is the codon ID and the value is a
     // ConditionCollection containing all conditions.
     public Hashtable Conditions {
       get;
     }

     // returns an ArrayList with all the codons defined
     // in this extension.
     public ArrayList CodonCollection {
       get;
       set;
     }
  }
}
```

The add-in class is not only a container class, it also performs important tasks such as filling the fields of a codon with values obtained from the xml attributes. This topic will be discussed in the next section.
Normally we will not be exposed to the `AddIn` or `Extension` class. This in fact is only useful for some sort of add-in manager which is capable of performing on the fly loading/unloading of add-ins or downloading new add-ins from the web or updating existing add-ins or similar tasks. But it is important to have looked at this class to know what is going on behind the scenes.

3.2 Overview of the Core.AddIns.Codons namespace



3.2.1 Definition of the ICodon interface

The basic codon interface which all codons must implement is the following:

```
public interface ICodon
{
  // returns the add-in in which this codon object was declared
  AddIn AddIn {
    get;
    set;
  }

  // returns the name of the xml node of this codon.
  string Name {
    get;
  }

  // returns the ID of this codon object.
  string ID {
    get;
    set;
  }
```

```
  // returns the Class which is used in the action corresponding
  // to this codon (may return null, if no action for this
  // codon is given)
  string Class {
    get;
    set;
  }

  // Insert this codon after the codons defined in this string
  string InsertAfter {
    get;
    set;
  }

  // Insert this codon before the codons defined in this string
  string InsertBefore {
    get;
    set;
  }

  // Creates an item with the specified sub items. And the current
  // Condition status for this item.
  object BuildItem(object caller, ArrayList subItems,
                   ConditionFailedAction action);
}
```

To simplify the task of the codon implementors, an abstract class `AbstractCodon` is defined, implementing all of the properties. The `AddIn` property is set after the construction of the codon by the `CodonBuilder`. All codons must have an empty constructor, as this is important for the construction process.

3.2.2 Defining new codons.

To define a new codon, we have to extend our codon class from the `AbstractCodon` class (due to limitations in the current reflection API, extending from ICodon is not sufficient) . This may change in later versions, so that extending from `ICodon` may be enough, but `AbstractCodon` will always be a valid base class.

Additionally, we have to specify a `CodonNameAttribute` initialized with the name of our codon (this will be the name of the Xml Node in our xml definition).

```
[CodonNameAttribute("MyCodon")]
public class MyCodonCodon : AbstractCodon
{
}
```

This will create a codon which may be used as xml tag `<MyCodon/>` having

the following attributes :

name	description	notes
id	the unique ID of the codon object	required attribute
class	the action class for this codon	
insertafter	codon IDs which the codon is inserted after	separated by ,
insertbefore	codon IDs which the codon is inserted before	separated by ,

Defining the attributes of a codon

This codon will have all properties as defined in the `ICodon` interface. To extend the attributes a codon may have, we have to define our own member variables using a `XmlMemberAttributeflagAttribute`.

```
[CodonNameAttribute("MyCodon")]
public class MyCodonCodon : AbstractCodon
{
      [XmlMemberAttributeflagAttribute("testMe", IsRequired=true)]
      string testMe = null;

      [XmlMemberAttributeflagAttribute("testMeToo")]
      string testMeToo = null;

      // canonical properties :)
      public string TestMe {
         get;
         set;
      }
      public string TestMeToo {
         get;
         set;
      }
}
```

Note that the `XmlMemberAttributeflagAttribute` has an `IsRequired` property which specifies whether the attribute is required for the node.

The convention is to define private attribute variables which will be shown to the outside by public properties which define get and set. Any other action related to the codons should be performed in their condition class or be defined in their creators.

Valid attributes may only be base types and enumerations, all other base types are not valid for the attribute tag.

With the `XmlMemberArrayAttribute` attribute we can specify an array as an xml attribute variable. The xml attribute string will be splitted on one or more characters and the resulting array will be written to the attribute.

```
[CodonNameAttribute("ArrayTest")]
public class ArrayTestCodon : AbstractCodon
{
      [XmlMemberAttributeflagAttribute("baseArray")]
      string[] baseArray = null;

      [XmlMemberAttributeflagAttribute("userArray",
       Separator = new char[] {'.', '/'})]
      string[] userArray = null;

      // canonical properties :)
      public string[] BaseArray {
         get;
         set;
      }
      public string[] UserArray {
         get;
         set;
      }
}
```

Now we have the following XmlNode in an extension tag :

```
<ArrayTest id = "Test"
           baseArray="a,b,c,d.e.f/g/h,i"
           userArray="a,b,c,d.e.f/g/h,i"/>
```

This will result in an `ArrayTestCodon` object in which `BaseArray` equals {"a", "b", "c", "d.e.f/g/h", "i" } and the `UserArray` equals {"a,b,c,d", "e", "f", "g", "h,i" }. The default separator character is the ','. Specifying the `Separator` character property of the `XmlMemberArrayAttribute` we can overwrite the default separator with new values (note that only character separators are supported).
Due to limitations in the .NET reflection API, currently only string arrays are supported. This may change in future versions.

Defining how an item should be built

The `BuildItem` method builds a single object with the given subItems and the current `ConditionFailedAction`.

Example: For a MenuItem codon it will return the `Windows.Forms.MenuItem` with a submenu created from the `subItems` ArrayList and depending on the `ConditionFailedAction` the item will be enabled or disabled.

```
public override object BuildItem(object caller,
```

```
                                ArrayList subItems,
                                ConditionFailedAction action)
{
  MenuItem newItem = null;

  if (subItems == null || subItems.Count == 0) {
    if (Class != null) {
      newItem = new MenuItem(Label, new EventHandler(
                                    new MenuEventHandler(caller,
                (IMenuCommand)AddIn.CreateObject(Class)).Execute));
    } else {
      newItem = new MenuItem(Label);
    }
  } else {
    newItem = new MenuItem(Label,
                 (MenuItem[])subItems.ToArray(typeof(MenuItem)));
  }

  newItem.Enabled = action != ConditionFailedAction.Disable;
  return newItem;
}

// internal class of the MenuItem codon
class MenuEventHandler
{
  IMenuCommand action;

  public MenuEventHandler(object caller, IMenuCommand action)
  {
    this.action        = action;
    this.action.Caller = caller;
  }

  public void Execute(object sender, EventArgs e)
  {
    action.Run();
  }
}
```

Note: `ConditionFailedAction.Exclude` **will never occur, because excluded items will not be built.**

The `IMenuCommand` **is simply a command with an object** `Caller` **property. All menu action classes must implement the** `IMenuCommand` **interface. Other codons may need other interfaces to be implemented by their specific classes (therefore the** `class` **attribute does not have a basic interface for all classes which might be specified with it). The documentation for which types the class attribute of a codon requires is part of the codon documentation.**

3.2.3 How codons are created.

Three classes are used in the codon creation process :

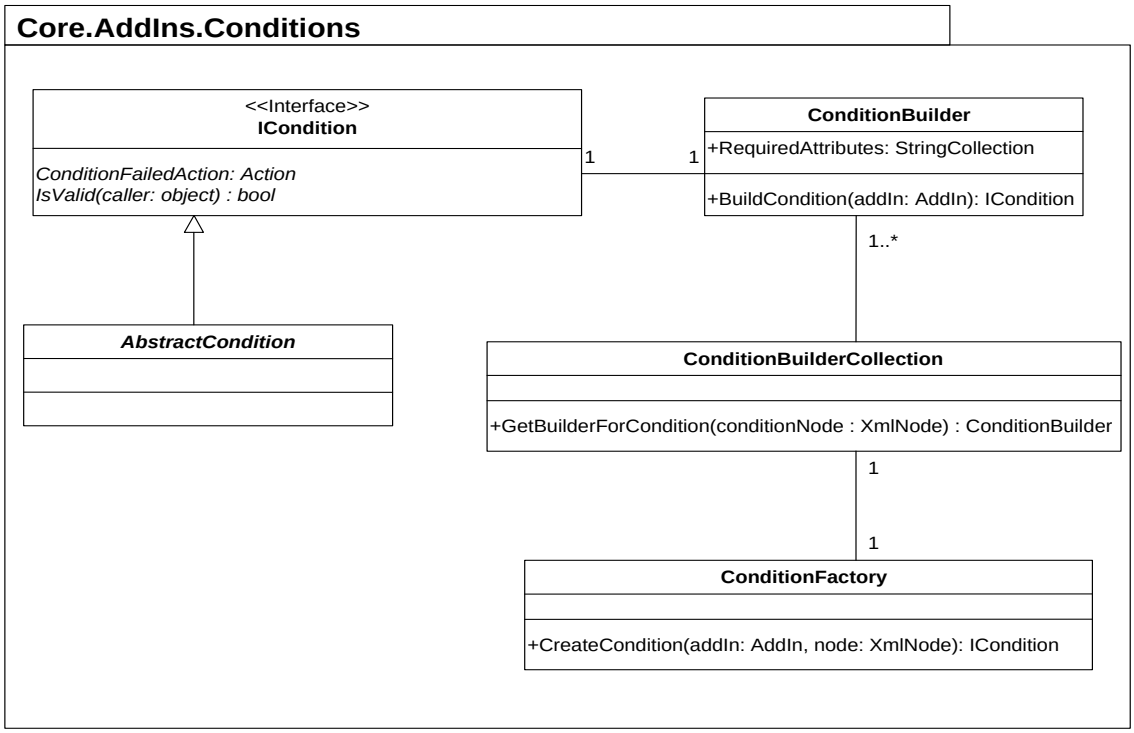
CodonBuilder **Builds an object of a specific class.**

CodonBuilderCollection **Contains all CodonBuilders and returns a specific** `CodonBuilder` **object for a given codon Name.**

CodonFactory **Can create a codon object out of a** `XmlNode`**. It uses a CodonBuilderCollection for determining the correct builder.**

Each codon must have an unique name (for example `MenuItem` **or** `CompressionStrategy`**). When an add-in xml is loaded, all specified runtime assemblies are searched for codons. If a codon class is found, a** `CodonBuilder` **will be created for this class and inserted into the** `CodonBuilderCollection` **of the** `CodonFactory` **object the global** `AddInTree` **object owns.**
Every time a codon object is to be created, the factory is called which will get the correct codon builder which actually builds the codon.
If this method proves to be a performance bottleneck, it will be replaced with the following: At the first start, or every time a new add-in is installed, all assemblies are scanned for codon definitions and for each codon definition a proxy class will be generated and written to a *codon definition* **assembly. Each proxy class can genenerate the original codon. The proxy class knows in which assembly the codon class is defined and also its name. When the add-in definition xml files are read for each extension node a proxy is generated which obtains a reference to the plain xml node from the file. Only when the tree path is read and a proxy is found at the tree node, the proxy will generate the codon and replaces itself with the codon in the tree.**
This would result in an on demand loading strategy which won't break existing add-ins or programs as the client building the item or getting a tree path will not be aware of the proxy having been added transparently. Note: Currently the basic loading algorithm is fast enough, therefore the proxy is currently not implemented.

3.3 Overview of the Core.AddIns.Conditions namespace



The `conditions` are built symmetrically to the codons, because they need the same high degree of extensibility. However they have a slightly different semantic.

3.3.1 Definition of the ICondition interface

The ICondition interface defines the basic functionality of a condition.

```
public interface ICondition
{
  // Returns the action which occurs, when this condition fails.
  ConditionFailedAction Action {
    get;
    set;
  }

  // Returns true, when the condition is valid otherwise false.
  bool IsValid(object caller);
}
```

As for codons, there exists a base class called `AbstractCondition` which is the base class for all conditions. The `IsValid` method returns the validity state of the condition. Thsi state may be dependent on the calling object. Which

calling objects are legal for a condition is part of the respective condition documentation. It may not require some specific object[3].

3.3.2 Condition actions

```
// Default actions, when a condition is failed.
public enum ConditionFailedAction {
        Nothing,
        Exclude,
        Disable
}
```

In case the user does not want to discard the codon when a condition fails, a GUI object — for example a button — might just be disabled when the corresponding textbox is empty. Therefore conditions have another attribute: `action`**. Currently, the following actions exist:**

ConditionFailedAction.Nothing **Does nothing, included only for completeness's sake.**

ConditionFailedAction.Exclude **Excludes the codon. (Default)**

ConditionFailedAction.Disable **Disables the codon. (Shows the object built, but deactivates it)**

3.3.3 Defining new conditions

We need to tag the conditions with a `ConditionAttribute` **and implement the** `IsValid` **method. All that was said about defining member attributes for codons applies to conditions as well.**

Example of a simple condition definition :

```
[ConditionAttribute()]
public class CompareStringsCondition : AbstractCondition
{
  [XmlMemberAttributeAttribute("string1", IsRequired=true)]
  string s1;

  [XmlMemberAttributeAttribute("string2", IsRequired=true)]
  string s2;

  // canonical properties :)
  public string String1 {
    get;
    set;
  }
  public string String2 {
```

[3]The behaviour is part of the specification of the condition. A condition may refer to the caller object, but various conditions also work fine without referral.

```
    get;
    set;
  }

  public override bool IsValid(object caller)
  {
     return s1 == s2;
  }
}
```

The `IsValid` method will be called every time an attempt is made to create the codons inside the `Conditional` tag of the condition.
This may be in case of a special event which *may* change some conditions ('dynamic case') or only once during the program run('static case'), depending on the codon and the client code. Generally, the implementation should be for the dynamic case .

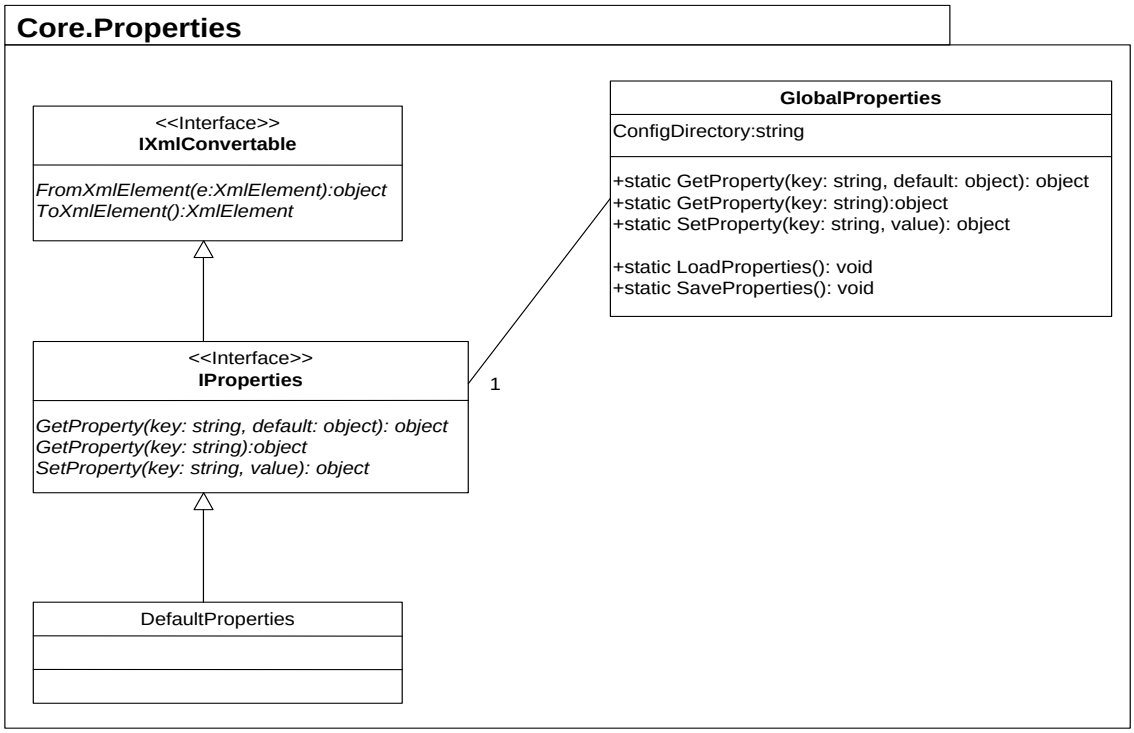
The given `caller` object is the object calling the creator. It is useful for determining conditions depending on the state of the caller. For example, the calling object may be used to determine the state of a dialog and set the IsValid value depending on its state.

4 Properties overview

4.1 The problem of property persistence

In almost all GUI based applications we confront the issue of persistence. This is not about the persistence of the data the application handles, but rather the persistence of the application's options, or states. The user will want to have the GUI state saved (when he opens the application again it should look the same as the last time he worked with it) and many program options need to be saved. When a program contains a plugin architecture, plugins need a way to save their properties as well. Implementing an individual property handler for each property in mid to large scale applications is not viable. Generally, the properties must work in a multi user environment. When the user starts the program for the first time, default values need to be loaded and then the program will be configured according to his needs. He certainly would not want another user to mess around with his personal configuration.
Thus, a common way of saving properties had to bes be introduced.

4.2 Overview of the Core.Properties namespace



In this namespace, all classes necessary for property handling are defined.

4.3 Definition of the IXmlConvertable interface

```
public interface IXmlConvertable
{
  // convert xml to an object
  object FromXmlElement(XmlElement element);

  // convert an object to xml
  XmlElement ToXmlElement(XmlDocument doc);
}
```

This interface is used to store the state of objects to the property structure which can not be converted to string and back. (This is the case for almost any non base types.)

We might have used some sort of serializing strategy using reflection to accomplish the task of storing an object to xml (and converting it back). This approach was choosen instead, as for most properties the base types should be sufficient— and more importantly — to maintain backwards compatibility. Serialization will not work if we change (or completely rewrite)

the internal structure of an object. Enforcing a user defineable loading and storing procedure can prevent problems with backwards compatibility.

4.3.1 Definition of the IProperties interface

```
public interface IProperties : IXmlConvertable
{
  // default property setter
  void SetProperty(string key, object val);

  // default property getter
  object GetProperty(string key, object defaultvalue);
  object GetProperty(string key);

  // overloaded property getter
  int         GetProperty(string key, int defaultvalue);
  bool        GetProperty(string key, bool defaultvalue);
  short       GetProperty(string key, short defaultvalue);
  byte        GetProperty(string key, byte defaultvalue);
  string      GetProperty(string key, string defaultvalue);
  System.Enum GetProperty(string key, System.Enum defaultvalue);

  // Returns a new cloned instance of IProperties
  IProperties Clone();

  // The property changed event handler is called
  // when a property has changed.
  event PropertyEventHandler PropertyChanged;
}
```

This is the base interface for a property class. With the get methods we either get the property saved under the key, or the specified default value. It is recommended to specify a default value, as the overloaded getters will not need user casts and the application is more stable (as no `null` objects will be returned)

4.3.2 Definition of the GlobalProperties class

```
public class GlobalProperties
{
  // returns the path of the default application
  // configuration directory
  public static string ConfigDirectory {
    get;
  }

  // methods for restoring & storing the global properties
  public static void LoadProperties();
  public static void SaveProperties();

  // static default property setter
```

```
  public static void SetProperty(string key, object val);

  // static default property getter
  public static object GetProperty(string key,
                                   object defaultvalue);
  public static object GetProperty(string key);

  // static overloaded property getter
  public static int         GetProperty(string key,
                                        int defaultvalue);
  public static bool        GetProperty(string key,
                                        bool defaultvalue);
  public static short       GetProperty(string key,
                                        short defaultvalue);
  public static byte        GetProperty(string key,
                                        byte defaultvalue);
  public static string      GetProperty(string key,
                                        string defaultvalue);
  public static System.Enum GetProperty(string key,
                                        System.Enum defaultvalue);

  // The static global property changed event handler, it is
  // called when a global property has changed.
  public static event PropertyEventHandler GlobalPropertyChanged;
}
```

Introduction

The `GlobalProperties` class basically implements the `IProperties` as static methods. It uses a single `DefaultProperties` object to store the property information.

Loading and storing

With a call to the static method `LoadProperties`, the global properties are loaded. With a call to the static method `SaveProperties`, the global properties are saved.

But what is going on behind the scenes? In the `GlobalProperties` class, some private constant fields are defined:

defaultPropertyDirectory This is the path of the default directory in which a default property file should exist.

propertyFileName This is the file name of the property file which is loaded.

configDirectory This is the directory of the current user's application. This is the only property which is made public through the `ConfigDirectory` property;

Firstly, the file `propertyFileName` in the `configDirectory` is loaded. When this fails, the Core attempts to load it from the `defaultPropertyDirectory`. In case this also fails, a `PropertyFileLoadException` is thrown.

4.3.3 Definition of the DefaultProperties class

The `DefaultProperties` class implements the `IProperties` interface. We use this class to give our own objects the capability of using properties. However, we only use it when an object needs its own property object. For example: When we write a text editor, each buffer may have its own property object which is cloned from a global default property object and when a user changes a property of one of these buffers, the others will not change the respective property.

4.3.4 Event handling

```
class PropertyEventArgs : EventArgs
{
  // returns the changed property object
  public IProperties Properties {
    get;
  }

  // The key of the changed property
  public string Key {
    get;
  }

  // The new value of the property
  public object NewValue {
    get;
  }

  // The old value of the property
  public object OldValue {
    get;
  }
}
```

This class is passed to the event handler which informs clients of a property change. With the new value and old value information, the client can determine what has changed and in which `IProperty` object this change occurred. Note : With the eventhandler we may get the internal `DefaultProperties` object in the `GlobalProperties` class but it should be treated as immutable.

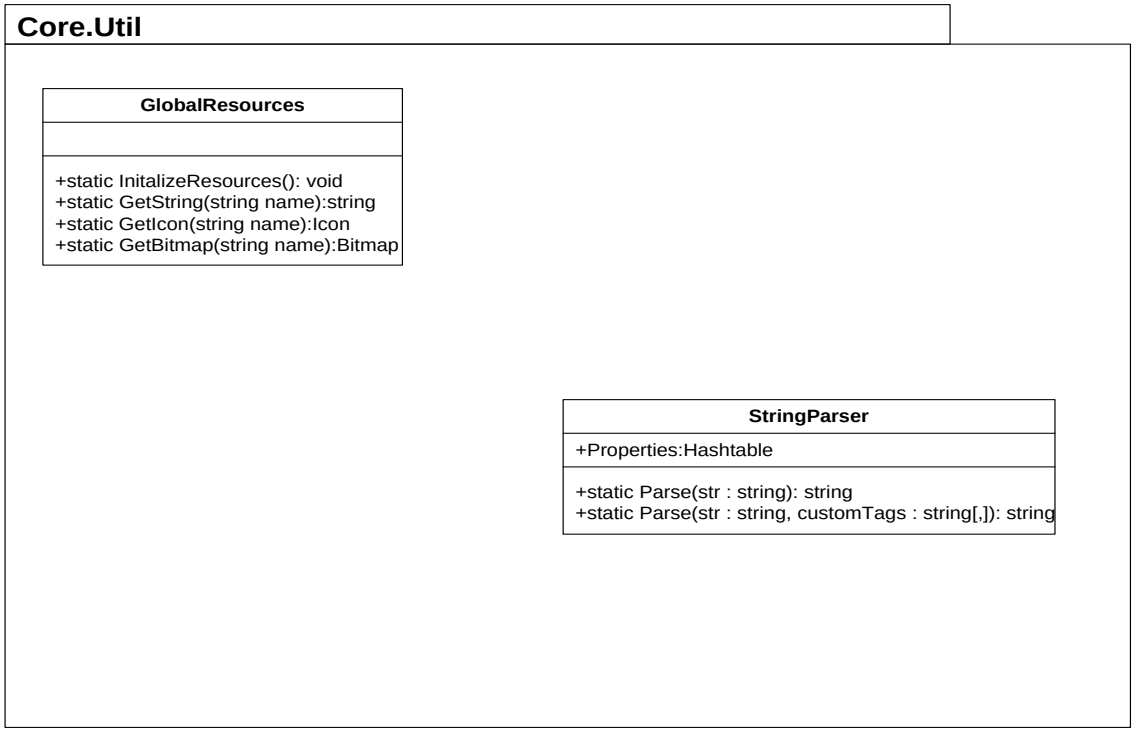
4.4 The Xml file format

```
<?xml version="1.0"?>
```

```
<NCvsProperties fileversion="1.0">
  <Properties>
    <Property key="CoreProperties.UILanguage" value="de-DE" />
    <Property key="NCvs.Connection.UseStreamComression" value="True" />

    <XmlConvertableProperty key="NCvs.Workspace.WorkspaceMemento">
      <WindowState bounds="34,99,1148,671" formwindowstate="Maximized"
                   fullscreen="False" />
    </XmlConvertableProperty>

  </Properties>
</NCvsProperties>
```

In this xml file we see that the root node has only one child: the global property node. This node contains all properties which were stored before the `SaveProperties()` method was called. The `Property` nodes contain the key and the base type values. When the property file is read, the key and value will be saved in a hashtable. In the `XmlConvertableProperty` node an XmlConvertable object is stored. The property reader can not get the type at runtime (the assembly may even not have been loaded when the property file is read). Instead, the complete XmlNode is stored in the hashtable and it is converted back to the object by using the type of the given defaultValue in the GetProperty method. If it is not read (and converted back), the node will be written back to the xml unchanged when saving the properties again. Therefore, the `object GetProperty(string key)` method is not to be used when getting an `IXmlConvertable` object the first time. (It is recommended to only use default values, thus preventing `null` errors and `IXmlConvertable` cast errors)
Note: When the `FromXmlElement` method of the `IXmlConvertable` interface does not return a new object, the state of the given default value object may have changed. Never assume a given default value state after a get to be the same as before the get.

5 Useful Core utilities



5.1 Localization

Modern applications need the capability of using localized resources. In the `Core.Util` namespace the `GlobalResources` class is defined which deals with the localization of resources and handles image resources (for icons).

```
public class GlobalResources
{
  // initializes the resource manager, done by the core system
  public static void InitializeResources();

  // returns a localized string
  public static string GetString(string name);

  // returns Icons or Bitmap object
```

```
  public static Icon   GetIcon(string name);
  public static Bitmap GetBitmap(string name);
}
```

When the GlobalResources class is initialized, it first loads the default resources and then the localized resources. On the first start, it obtains the localized resources using the current UI culture. The language string is saved in the GlobalProperties under `CoreProperties.UILanguage` — this string is used the next time the application loads. The application may want to make the UI language user selectable instead of auto setting the culture.
The get methods return the object by name preferably from the local resources. If it does not exist in the localized resources (or the culture doesn't have resources defined), the value from the default resources is returned.
An exception is thrown when the resource string was not found.

5.2 Putting the GlobalResources and the rest together.

You may have noticed that the global resources are 'stand alone'. To define an interface for the user, the `StringParser` class was defined in the `Core.Util` namespace.

```
public class StringParser
{
  public static PropertyDictionary Properties {
    get;
  }
  public static string Parse(string str);
  public static string Parse(string input,
                             string[,] customProperties);
}
```

With the `StringParser.Parse` method we can expand **${[MACRONAME]}** macros. We can get all environment variables with the StringParser using `env.[NAME]` where `[NAME]` is the name of the environment variable.

Other macros are `DATE` and `TIME`. We can expand resources using `RES:[RESOURCENAME]` as macroname with `PROPERTY:[PROPERTYNAME]` expanding to a property value (the string value of the property).

By using the `Parse(string input, string[,] customProperties);` method we can define additional macros by specifing customProperties as a tuple of strings, where the first string is the property name and the second the value.

The `StringParser` is a small but important class. It is used to localize resource strings in codon attributes and generally all strings which are visible to the user will be a run through the `Parse` method of the StringParser.