# A Component Plug-in Architecture for the .NET Platform[1]

Reinhard Wolfinger, Deepak Dhungana
Herbert Prähofer, Hanspeter Mössenböck

Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University, 4040 Linz, Austria

{wolfinger,dhungana,praehofer,moessenboeck}@ase.jku.at

**Abstract.** Plug-in architectures and platforms represent a promising approach for building software systems which are extensible and customizable to the particular needs of the individual user. For example, the Eclipse platform, as the most prominent representative of plug-in systems, is based on a unique plug-in and extensibility concept and has succeeded in establishing itself as the leading platform for the development of tool environments. This paper introduces a new plug-in architecture for the .NET platform which shows much resemblance to Eclipse. However, whereas Eclipse is a Java-based system and uses XML to describe extensions, our architecture relies on .NET concepts such as custom attributes and metadata to specify relevant information directly in the source code of an application. We argue that this approach is more readable and easier to maintain. As a case study for our plug-in architecture we present a new plug-in platform for implementation of rich client applications in .NET.

## 1 Introduction

Originally made popular by Web browsers, plug-in platforms enable the extension of a core application with new features implemented as components that are plugged into the core at load time or even at run time and integrate seamlessly with it. Feature-bloated applications like Microsoft Word evidence what happens if a monolithic application follows the *one-size-fits-all* approach. Microsoft continues to receive user feedback with requests for features that already exist in the product. Typical users struggle to find their 10%-share of the feature set that they really want to use. The plug-in approach on the other hand strives for compact application cores that can be extended with plug-in components tailored to the users' needs. It improves focus and reduces clutter by providing a customized user environment.

Eclipse [7] is certainly the most prominent representative of those plug-in platforms and has driven the idea to its extreme: “Everything is a plug-in!” [1]. More-

[1] This work has been conducted in cooperation with BMD Systemhaus GmbH, Austria, and has been supported by the Christian Doppler Forschungsgesellschaft, Austria.

over, other interesting approaches exist (see related work below). For example, OSGi [16] is a Java-based technology for deploying and managing coarse-grained components. It also serves as the deployment technique for Eclipse plug-ins. Mozilla [13] represents a further interesting plug-in platform where user interface contributions are defined in the declarative language XUL. Despite their diverse technological foundations, all those systems have in common the focus on extensibility of applications.

In addition to facilitating extensibility of applications, plug-in architectures and platforms represent an interesting and promising approach for providing reusable building blocks. Assembling systems from pre-fabricated building blocks has been regarded as an appealing approach to software construction since the very early days of software engineering [11]. Originally, object-oriented technology was supposed to make systematic development of reusable building blocks and wide-scale reuse feasible [2]. Today, it is generally agreed that object-oriented technology as such has not fulfilled those expectations [20]. Although we have seen much progress into this direction in the last decades [19], [3], component-based software engineering still has not reached a level of maturity that is taken for granted in other engineering disciplines.

We argue that plug-in approaches represent a significant progress for making a component-based software development reality. So it seems that Eclipse has succeeded where previous approaches have failed, namely in establishing a real component market. Today a huge community of developers and software vendors has committed itself to Eclipse as the technological basis for developing reusable components and thousands of Eclipse plug-ins can be found on the Web.

The success of Eclipse and similar systems has many reasons, some of which go far beyond pure technical considerations. However, there are several technical features which have contributed to the success of plug-in systems:

- Plug-in components are coarse-grained, i.e., they are like small applications with features which are of direct value for the user. In that, they are mainly self-contained and have limited dependencies on other components.
- There are clear rules on how to specify dependencies and interactions between components. This results in precise documentation on how systems can be extended and how plug-ins shall interact.
- Eclipse and similar systems have demonstrated ways how plug-in components can be integrated seamlessly into working environments. Working environments can grow in a disciplined and determined manner allowing the users to create their individual working environments by selecting from a set of plug-ins.

This paper introduces a new plug-in architecture for the .NET platform which shows much resemblance to Eclipse. However, whereas Eclipse describes extension points and extensions with dedicated XML configuration files, our architecture relies on .NET concepts such as custom attributes and metadata to specify relevant information directly in the source code of an application. We argue that this approach is more readable and easier to maintain. Moreover, it exploits .NET specific features for plug-in deployment and discovery. As a case study for our plug-in architecture we present a new plug-in platform for rich client applications called CAP.NET. We describe its design as well as a prototypical implementation and show how it supports users assembling their individual working environments.

## 1.1 Related Work

One of the first runtime extensible systems was Emacs ("Editor MACroS") [18]. Emacs extensions are written in *elisp* – a Lisp dialect – and installed by setting paths in an initialization file. In that, Emacs can be described as a readily extensible scripting framework, to which new scripts can be added at runtime.

Many Web browsers make use of the “plugging” concept. Mozilla [13], for example, lets developers define new extensions and makes use of various technologies for this purpose. In Mozilla the user interface is represented by an XML data model. Plug-ins specify their user interface contributions in the declarative language XUL (another XML language) and with the help of JavaScript and XPCOM one can add dynamic behaviour to the UI elements. XPConnect bridges XPCOM [17] and JavaScript by allowing JavaScript to access and manipulate XPCOM objects. Applications built with this technology are not limited to the Mozilla Web browser but range from different Web browsers (FireFox, Mozilla Suite, Netscape), email clients (Thunderbird), calendar applications (Sunbird) to integrated development environments (Kommodo) and Web-design applications (NVU).

OSGi [16] is a Java-based technology for deploying and managing coarse-grained components. The Open Service Gateway Initiative (OSGi) defines several mechanisms that are relevant for a plug-in framework. Lifecycle management of components (referred to as bundles in OSGi) and hot update are possible using OSGi. Additionally, OSGi offers a service concept and a set of service standards for component integration and interaction. Technically, the OSGi service framework can be boiled down to a custom, dynamic Java class loader and a service registry that is globally accessible within a single Java virtual machine [9].

The Eclipse Platform [7] is certainly the most prominent plug-in platform today. Eclipse is built upon a small core and all further functionality is provided by a (usually huge) set of plug-ins. Plug-ins for Eclipse are written in Java and are delivered as JAR libraries. Plug-ins declare their interconnections to other plug-ins using a manifest in XML format. The idea is quite simple: a plug-in declares named extension points and extensions to extension points in other plug-ins. The platform matches extensions with their corresponding extension point declarations by name, discovers plug-in dependencies in this way, and integrates the plug-ins to a comprehensive working environment at start-up without actually loading the code. The resulting plug-in registry is available via the platform API. Any problems, such as extensions to missing extension points, are detected and logged. Eclipse evolved as it moved from version 2.x to 3.0. In version 3.0.7 it adopted the OSGi Service Platform (SP) as a foundation for plug-in management. Backed with OSGi it allows hot updates of plug-ins, i.e., updating the code and reloading a plug-in while the Eclipse system keeps running.

NetBeans [15], as the main competitor of the Eclipse Java development environment, has introduced a plug-in concept which differs significantly from that in

Eclipse. In NetBeans, plug-ins are referred to as modules and, as in Eclipse, are JAR libraries contained in a plug-in directory within the NetBeans environment. However, extension mechanisms and plug-in integration are based on a so-called *virtual file system* that, in essence, represents the hierarchical structure of the application. Plug-ins define their extensions in XML documents (called *layer.xml*) by specifying their contributions to the virtual file system. For example, a plug-in which would like to add a menu item "Format" in sub menu "Source" would simply specify that it has a contribution to the virtual directory path "Menu/Source" with name "Format".

### 1.2 Outline

This paper is structured as follows: Section 2 defines basic terms and concepts and derives requirements for a plug-in architecture and a plug-in platform. In Section 3 we will discuss some .NET features which have been important in our approach. Section 4 introduces the basic concepts of our plug-in architecture, namely extension, deployment, and discovery mechanisms. In Section 5 we validate our approach by a prototypical implementation of a rich client platform. Finally Section 6 discusses our approach and achievements and gives an outlook to future work.

## 2 Terms and Concepts

### 2.1 Basic Mechanisms of a Plug-in Platform

A plug-in platform enables components to plug into an application core at load time or at run time. For that purpose the platform requires the following basic mechanisms:

*(a) Plugging.* An essential principle of a plug-in architecture is that system extensions are carried out in controlled, restricted and determined manner. Therefore, in a plug-in architecture there have to be means which allow specifying explicitly how a component should be extended and how other plug-in components make their contributions. In Eclipse, for example, this is done by *extension point* and *extension* specifications in XML. We adopt a notion of extension *slot*, i.e., the specification how an extension should occur, and the *extension*, i.e., the specification how a plug-in makes a contribution to a particular slot. System integration, i.e., combining the set of plug-ins at hand into a integrated running system, is solely accomplished by reading and exploiting this information about extension slots and fitting extensions.

*(b) Deployment.* Plug-in components are like small applications that extend the application core by new services. Nontrivial services may consist of multiple extensions plugging into different slots defined by one or several extension hosts. Extensions that belong together are merged into a single plug-in component that serves as a single unit of deployment and versioning.

*(c) Discovery.* Safe and easy plug-in activation should not require complicated and error-prone configuration tasks. Therefore, plug-ins are simply moved to a central place called a *plug-in repository* where they are discovered and activated automatically at application load or run time. Accordingly, removing a plug-in from the repository deactivates it.

## 2.2 Slots and Extensions

As discussed above, a plug-in is a deployable software unit which has explicit specifications of its slots and extensions. Slot specifications define how other plug-ins are intended to extend the functionality of this plug-in, whereas extension specifications define how this plug-in makes contributions to slots of others. Therefore, slots and extension specifications have to match. In essence, slots declare the types of information a plug-in expects and the extensions fill this information slots accordingly. In its simplest form, a plug-in specification is a structured list of name/value-pairs where the slot specifies the required names and value ranges and the extension specification defines appropriate values for the extension at hand.

The component defining the slot is called the *extension host* and the component implementing the extension is called the *extension contributor*. Extension contributors again can define their own slots where other plug-ins can contribute allowing the whole system to grow. A non-trivial plug-in can contain multiple extensions plugging into different slots.

Usually, plug-in extensions will occur on the level of run-time behavior, i.e., plug-in host and contributor will communicate based on a defined protocol in order to accomplish a particular task. The collaboration between the host and its contributor is defined in the form of required and provided interfaces. The host will define the required interface and the extension contributor has to provide an implementation for it.

Fig. 1 depicts the structure of slot and extension specifications in host and contributor plug-ins. The interface in the host and the implementation class in the contributor specify the agreed collaboration protocol. Additional name/value pairs define other properties that the host requires to make use of the extension.

## 2.3 Further Services of a Plug-in Platform

A plug-in platform built on the basic mechanisms presented above can also provide the following services:

*(a) Hot Plugging.* Having to restart an application in order to install new components leads to annoying interrupts of the user's work flow and should be avoided. Hot plugging means the ability to add, update and remove plug-ins while the application is running.

*(b) Auto-Update.* Patches are a common way of supplying small updates to pieces of software in order to update it or to fix problems. If the update process is automated

users are relieved from this tedious and error-prone task. A plug-in platform can therefore periodically scan the plug-in repository and compare version information with an installation repository. If newer versions of plug-ins are available, they are copied from the installation to the plug-in repository and are reactivated when the application is restarted. If hot plugging is supported, no restart is required.

*(c) Sandboxing*. Malicious or unreliable plug-ins represent a security hazard for the application. A sandbox is a secure environment for safely running plug-ins within well-defined limitations to their possible set of actions.

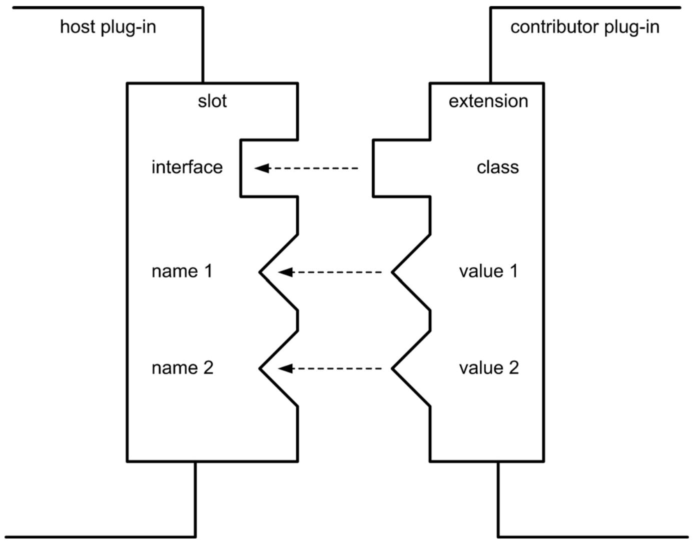

**Fig. 1.** Slot and extension in host and contributor plug-in

## 3 .NET Framework Concepts

The .NET Framework offers advanced concepts which form a technological basis for the plug-in approach presented. Those are .NET custom attributes, assemblies, meta-data and reflection. We shortly discuss those topics now.

*(a) Custom attributes*

Custom attributes are pieces of meta-information that can be attached to language constructs such as classes, interfaces, methods or fields in the source code of an application. At run time the attributes attached to a language construct can be retrieved using reflection [12]. In addition to pre-defined attributes programmers can declare custom attributes by implementing attribute classes with arbitrary properties whose values can be set when the attribute is attached to a language construct.

Adding custom metadata that can be evaluated by development tools is a common usage scenario in .NET. A well-known example is the `WebMethod` attribute which indicates that a method is exposed as part of an XML Web service.

```
public class StockTicker : WebService {
  [WebMethod]
  public double GetQuote(string symbol) { ... }
}
```

The Web Services Description Language tool (wsdl.exe) is an example of a development tool that uses reflection to read out the `WebMethod` attribute for identifying a method as a Web method and, from this information, creating contract files or proxy code.

Our plug-in architecture makes similar use of attributes for representing information about slots or extensions in plug-in code (see Section 4.1).

*(b) Assemblies*

An assembly is the basic packaging unit in .NET. It is the smallest unit for loading, deployment, versioning and security. Assemblies can come as executables (*.exe) or as library components (*.dll). They contain metadata describing types, resources and referenced assemblies. Because assemblies are self-describing, assembly component deployment is as simple as an copy operation. There are no issues with class or type library registration as in traditional COM deployment. Lack of registry dependency and support for side-by-side component deployment avoids the problem known as "DLL hell" [10].

Strong name identification and assembly version information are used to identify components. An assembly version number is represented as a four-part number. For example, version 1.5.1254.0 indicates 1 as the major version, 5 as the minor version, 1254 as the build number, and 0 as the revision number. To give an example a version number is attached to the `StockTicker` program (see above) through the AssemblyVersion-Attribute.

```
[assembly: AssemblyVersion("1.5.1254.0")]
public class StockTicker { ... }
```

A component update service can acquire the component version using reflection like this:

```
Version v =
  AssemblName.GetAssemblyName("ticker.dll").Version;
```

In our plug-in architecture a dll assembly is used as a container for a plug-in component. Strong name identification and assembly version information are used to identify plug-ins and to facilitate auto-update.

*(c) Metadata.*

An assembly does not only store code but also metadata describing the symbolic information of all types, methods, fields and other entities in the assembly. An assembly's metadata is generated automatically by the compiler from the source code.

.NET makes it possible to retrieve the metadata of an assembly at run time using reflection [10]. The sample code below demonstrates how a tool can search the `StockTicker` class (see above) for methods with the `WebMethod` attribute attached. For all the methods in the `StockTicker` class it will retrieve all the attributes of type `WebMethodAttribute` using the reflection method `GetCustomAttributes`. If the length of the array returned is greater than 0, the first array element is accessed and casted to the `WebMethodAttribute` type.

```
foreach(MethodInfo mi in typeof(StockTicker).GetMethods()) {
  object[] webMethodAttrs = mi.GetCustomAttributes(
                         typeof(WebMethodAttribute), true);
  if(webMethodAttrs.Length > 0) {
    WebMethodAttribute webMethodAttr =
            (WebMethodAttribute) webMethodAttrs[0];
    // use WebMethodAttribute
  }
}
```

Our plug-in platform uses reflection for discovery. The discovery service scans the plug-in repository and activates extensions by reading their extension definition from metadata.

## 4 A Plug-in Architecture for .NET

In this section we will show how the .NET-specific concepts described in Section 3 can be used to implement the basic mechanisms of a plug-in platform as described in Section 2. In particular, we show

- how to define slots and extensions with .NET attributes,
- how to use .NET assemblies for plug-in packaging and deployment, and
- how to use reflection for plug-in discovery and activation.

We will showcase this approach with two sample extensions. The first example defines an extension slot for pluggable logging functionality. An extension contributor will receive logging information from the host and logs this information in its specific way. The second extended example introduces an additional custom property to differentiate various message types (e.g. error, warning, info).

### 4.1 Defining Slots and Extensions with .NET Attributes

Our specification of slots and extensions is based on .NET attributes. In Section 2 we have seen that a slot can specify one or more interfaces, which have to be implemented by the extension. In our first example the extension host defines an interface `ILog` and applies the attribute `Slot` with a name `"Log"` to the interface.

```
[Slot("Log")]
public interface ILog {
  void Write(string msg);
}
```

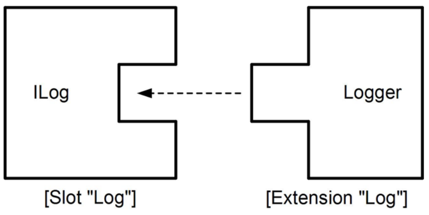

**Fig. 2.** Simple logger slot and extension

The `Slot` attribute is predefined by our plug-in platform. It is used to tag any program elements which belong to a particular slot. The slot is identified by a unique name. In the example above, the `Slot` attribute simply says that the `ILog` interface is an interface of the slot with the name `"Log"`. The following code shows the definition of the `Slot` attribute.

```
class SlotAttribute : Attribute {
  public SlotAttribute(string name) { ... }
  private string name;
}
```

The interface `ILog` declares a method `Write` which the host calls to log messages and which has to be implemented by contributor plug-ins. The sample extension `Logger` writes messages to the console. The class `Logger` provides an implementation for the required interface `ILog`.

```
[Extension("Log")]
public class Logger : ILog {
  public void Write(string msg) {
    Console.WriteLine(msg);
  }
}
```

Extensions in contributor plug-ins have to be tagged by the custom attribute `Exten-sion` which is also predefined in the platform. The same name as in the slot is used to uniquely identify the slot to be extended. The following code shows the definition of the `Extension` attribute.

```
class ExtensionAttribute : Attribute {
  public ExtensionAttribute(string name) { ... }
  private string name;
}
```

In the example above, the class `Logger` is now referred to as an extension of the slot `"Log"` because it is associated to this slot by the attribute `Extension`. Furthermore, it is a valid extension that conforms to the requirements of the `Slot` declaration as it implements the interface `ILog`.

## Logger with a Custom Property

The previous example is now extended to demonstrate the use of custom properties. In this example, we assume that the host allows the extension to choose between different message types which should be logged. For example, if the extension specifies `Info`, `Warning` or `Error` as its message types, the host will forward only the respective logging information.

The host defines a custom attribute class `MessageType`. The standard `AttributeUsage` attribute specifies that `MessageType` can only be attached to class definitions. The `Slot` attribute defines that `MessageType` is associated to a `"Log"` slot.

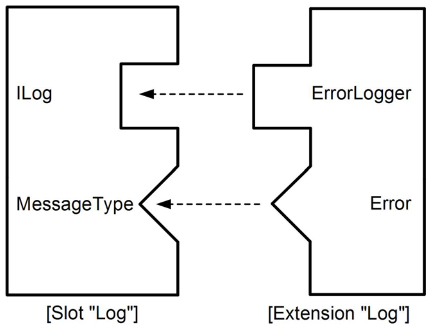

**Fig. 3.** Logger with custom property

```
public enum MessageTypeEnum {Info, Warning, Error}

[Slot("Log")]
[AttributeUsage(AttributeTargets.Class)]
public class MessageType : Attribute {
  public MessageType(MessageTypeEnum type) { ... }
  private MessageTypeEnum type;
}
```

A contributor plug-in can use this custom attribute in the extension implementation. In the following `ErrorLogger` implementation the `Slot` attribute is used to specify that this class is an extension to the `"Log"` slot and the `MessageType` attribute is used to specify that this logger implementation is intended to accept error logs only.

```
[Extension("Log")]
[MessageType(MessageTypeEnum.Error)]
public class ErrorLogger: ILog {
  public void Write(string msg){
    // do something
  }
}
```

In summary, specifying slots and extensions works as follows. In an extension host a slot is specified in the following way:

- There is a `Slot` attribute which is used to tag program elements of the host, i.e. interfaces and custom attribute classes, which belong to a particular slot that is identified by a unique name.
- The host will define one or several interfaces which are intended to be implemented by the extension contributors. They are marked with the `Slot` attribute.
- In most cases the host will also define one or several custom attribute classes which are intended to be used by the extension contributors to provide static information. That means, custom attributes are used to embody the name/value pairs. Again the `Slot` attribute is used to assign the attribute class to a particular slot.

In an extension contributor the extension is specified as follows:

- There is an `Extension` attribute that is used to tag the class of the contributor in order to make a contribution to a particular slot. Thereby, the unique slot identifier given in the `Slot` attribute is used.
- For a slot interface there is a class implementing that interface in the contributor. This class is tagged with the `Extension` attribute denoting that it is an extension of a particular slot.
- The custom attributes defined in the slot specification will be used in the contributor to provide the respective static extension information. They are also attached to the extending class.

## 4.2 Deployment and Update

The class that provides an extension is packed into a plug-in assembly for deployment. To prepare for automatic plug-in update, we need to add version information to the plug-in assembly. We continue with the error logger example from the previous section and add version information.

```
using System;
using System.Reflection;

[assembly: AssemblyVersion("0.1.*")]

[Extension("Log")]
[MessageType(MessageTypeEnum.Error)]
public class ErrorLogger : ILog {
  public void Write(string msg) {
    // do something
  }
}
```

Major version is 0, minor version is 1, and by specifying a wildcard for build and revision number, the compiler will insert adequate values. The following command builds the error logger plug-in.

```
csc /reference:platform.dll /target:library
/out:errorlogger.dll errorlogger.cs
```

The plug-in `errorlogger.dll` contains the extension and version information and is ready for deployment. Deployment means to move the plug-in into the repository. A repository is a folder in the file system that contains all active plug-ins for an extension host. When a plug-in is moved to the repository, the plug-in platform will automatically discover the newly installed plug-in and activate it (see Section 4.3).

Plug-in providers may provide updates with new features or fixes for problems. The Auto-Update service uses reflection to acquire the version info of plug-ins in the repository. It compares the version number of the currently installed plug-in and compares it to a installation repository that provides updated plug-ins. If updates are available, the update service replaces the plug-in in the repository with the newer version from the server.

```
Version v1 = AssemblyName.
  GetAssemblyName("errorlogger.dll").Version;
Version v2 = AssemblyName.
  GetAssemblyName("\\install\errorlogger.dll").Version;
if(v2 > v1) {
  // do update
}
```

The update process requires the plug-in platform to be restarted. An active plug-in means that its assembly is load in an application domain. As of .NET 2.0, assemblies cannot be individually unloaded. Consequently the update service shuts down the extension host, installs updated plug-in in the repository and restarts the extension host.

### 4.3 Discovery and Activation

At start-up the extension host searches the plug-in repository to discover plug-ins. It uses reflection to browse all classes in plug-in assemblies to look for `Extension` attributes. Classes with that attribute attached contain extensions. A conformance test checks if the extension provides the required interfaces and properties. Valid plug-ins are represented in the *plug-in registry*, which is a data structure that represents relations between slots and extensions, as well as inter-extension dependencies. The source code below shows a simplified discovery routine.

```
foreach(string filename in Directory.GetFiles(
        "plugins", "*.dll")) {
  Assembly a = Assembly.ReflectionOnlyLoadFrom(filename);
  foreach(Type t in a.GetTypes()) {
    object[] attrs = t.GetCustomAttributes(
                     typeof(ExtensionAttribute),true);
    if(attrs.Length > 0) {
      // check conformance
      // add it to the registry
    }
  }
}
```

Discovery uses the reflection-only context, which means that none of the plug-ins are actually loaded yet. Our plug-in platform supports lazy-loading, which means that extensions are loaded at the latest possible point in time. For example, when lazy-loading is applied to user interface elements, the plug-in is not loaded until the user performs an action on the user interface element and activates the respective function.

The code sample below shows how a plug-in is loaded and an extension is instantiated and used.

```
Assembly a = Assembly.LoadFrom("plugin\\errorlogger.dll");
ILog log = a.CreateInstance("ErrorLogger");
log.Write("Hello world!");
```

## 5 CAP.NET: A Rich Client Application Platform in .NET

CAP.NET (Client Application Platform .NET) is a platform for the realization of rich client applications in .NET and has been developed primarily for the validation of the plug-in architectural concepts as presented in Section 4. The idea of CAP.NET is to lay a basis for the realization of plug-in components for rich client workbenches, to allow the selection of an individual set of plug-in components by a user, and the integration of this set into a comprehensive and seamless user interface.

CAP.NET focuses on user interface integration, plug-in component deployment and life-cycle management. It is designed and built to meet the following requirements:

- integrate a variety of plug-ins for different tasks into a single rich client application
- facilitate seamless integration of user interface elements contributed by different plug-ins
- provide an update mechanism for plug-ins
- provide a framework for rapid application development of rich client components.

In addition to the plug-in mechanism presented in Section 4 CAP.NET provides the following features:

- a concrete plug-in discovery, deployment and update mechanism which uses a Web-Service interface on the download server
- a workbench component based on a well-defined user interface paradigm which provides several extension slots for plugging in user interface elements as plug-in components.

Fig. 4 shows the architecture of the CAP.NET platform. The platform core implements the plug-in discovery and start-up mechanisms and contains the plug-in registry. Additionally, the security component is responsible for managing rights and roles of plug-ins.

Everything else in CAP.NET is a plug-in. The core provides one extension slot `"cap.ui.workbench"`, which is intended to be filled by a workbench component. So far we have implemented just one kind of workbench component, but other workbench components following different UI paradigms could be implemented and plugged in as well. The realized `Workbench` component has several extension slots allowing further components to plug-in and in this way make contributions to the overall working environment, as we will see in Section 5.4. The update manager is also a plug-in and handles assembly deployment and update.

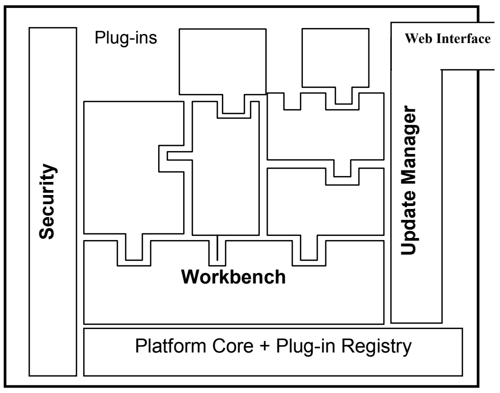

**Fig. 4.** Architecture of CAP.NET

In the following we will present CAP.NET in some detail. First, we will outline the platform core which implements the basic mechanisms. Then we will present the UI paradigm and the workbench components as well as the workbench extension slots and how the UI extensions are integrated. Finally, we will show some example plug-ins and an example user workbench.

## 5.1 Platform Core

At start-up, the platform core discovers the available plug-ins. It looks in the installation directory for files named `*.dll`. These plug-in assemblies are loaded using `ReflectionOnlyLoad` and scanned for slot and extension attributes.

The plug-ins discovered during this process are stored in the plug-in registry, which is basically a set of `Plugin` objects. The platform core is responsible for registering and managing plug-in assemblies and allows easy access to data and resources from the framework, including resources shared among plug-ins.

The plug-in registry simplifies the integration of extensions. A host component that defines a certain slot can find out if extensions to this slot are available using the plug-in registry which holds the static descriptions of the available extensions.

## 5.2 Update Manager

The update manager is also a plug-in. It periodically connects to an update Web Service running on an installation server. This Web Service checks whether newer versions of the installed plug-ins are available on the server and returns them to the update manager which installs them into the client's plug-in directory. This makes the update process extremely simple. The system administrator at the server side simply copies new versions of plug-ins into the server's plug-in directory. Any clients relying on these plug-ins detect the new versions automatically and copy them over.

## 5.3 User Interface Paradigm and Workbench Plug-in

User interface integration is about integrating different contributions into an overall user interface. To make this work, user interface integration has to be based on a general user interface paradigm, i.e. a general set-up and a general working principle that all applications obey. We have defined such a user interface paradigm for rich client applications. It focuses on the notion of *user tasks*, i.e., tasks that a user wants to work on. Depending on the chosen tasks the working environment will present itself in different ways. In general, a CAP.NET user interface consists of the following four panes: task navigation, task content, task commands, and views (see Fig. 5).

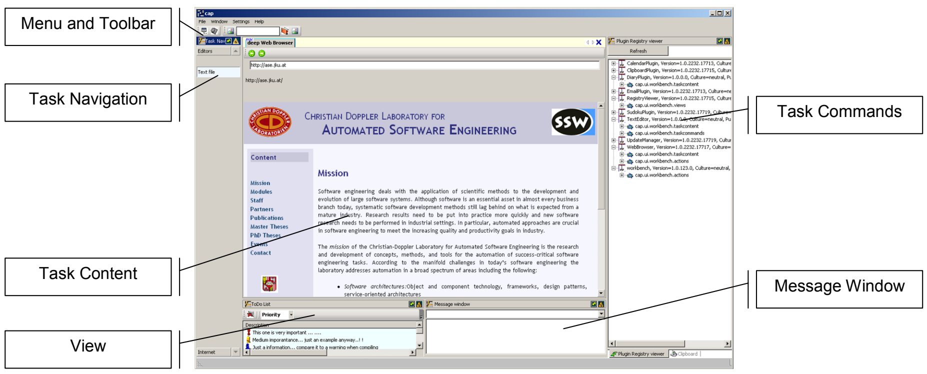

**Fig. 5.** CAP.NET Workbench

### *Task navigation*

This is a UI element which allows navigation between different tasks in a hierarchical manner and is usually placed on the left side of the working environment. It shows all available tasks in a hierarchical arrangement and is always visible.

### *Task content*

The task content window is the working window for a particular task. It will be either some sort of editor or an input form allowing the manipulation of data and objects. It can display the data graphically, textually, in forms etc. and can react on commands to change the data. Task content elements are based on the model-view-controller (MVC) pattern and allow the user to open, edit and save data objects. They follow an open-save-close life cycle much like file-system-based tools.

*Task commands*

It is common to have a set of commands for every task (e.g. a search command and a replace command for a text editor). These commands are displayed in task command windows, which are little control panes usually placed to the right of the task content window. There can be several task command windows for one task content.

*Views*

Views are UI elements which provide different views on a task's data. They can be used for navigation but not for changing data. For example, in a development environment there could be different views of the code that is being written. One view could display the variables and methods while another view could display properties and their values. A view may also augment other views by providing information about the currently selected object.

In addition to these special UI elements there are standard menus and toolbars as well as standard windows such as a message window or a to-do list. The `Workbench` plug-in realizes this UI paradigm and provides extension slots for tasks, commands, views and other elements, allowing custom plug-ins to make their UI contributions.

## 5.4 Workbench Extension Slots

In the following we outline the extension slots of the `Workbench` plug-in and show how contributor plug-ins can use them.

### **`cap.ui.workbench.actions`**

Additions to the menu bar and the toolbar are referred to as *actions*. This is because they represent some kind of user interaction, e.g. selecting a menu command or clicking a button in the toolbar.

In order to make a new menu or toolbar item available, the user has to provide code that has to be called, whenever the user clicks on that item. For that purpose an interface `IAction` has to be defined. This interface contains a method `OnClick`, which is called when the menu or toolbar item is clicked.

Since this slot can serve two purposes, i.e. the installation of a menu item or a toolbar item, there are two different attributes (`MenuAction` and `ToolBarAction`) that are used to provide the required information for UI integration. With the help of these attributes, the framework can extract the static information required for user interface integration without having to load the code.

The following code shows how a menu item is installed into the workbench. The class `UpdateMenuPlugin` extends the slot `cap.ui.workbench.actions`. The `MenuAction` attribute specifies the location where the new menu item should be inserted into the menus of the workbench.

```
[Extension("cap.ui.workbench.actions")]
[MenuAction(MenuPath = "Settings/Web")]
class UpdateMenuPlugin : IAction {
  public void OnClick(object sender, EventArgs args) { ... }
}
```

This will insert a menu item “Web” into the “Settings” menu. If the user selects this menu item the class `UpdateMenuPlugin` will be loaded and the `OnClick` method will be called.

### `cap.ui.workbench.taskcontent`

For contributing a new task content element there is the slot "`cap.ui.work-bench.taskcontent`" as well as the interface `ITaskContent` which defines methods for handling task content elements. Methods like `OnNew`, `OnSave` etc. are intended to be called when the respective actions on the content element are carried out. These operations apply to the currently active content. An extension to this slot has to implement the following interface `ITaskContent`.

```
[Slot("cap.ui.workbench.taskcontent")]
public interface ITaskContent{
  void OnNew();
  void OnSave();
  void OnSaveAs();
  void OnOpen(String filename);
  void OnClose(object sender, FormClosingEventArgs e);
  void OnTitleChanged(object sender, EventArgs e);
}
```

A `TaskContent` attribute can be used to provide information such as the file extension the task content is related to. For example, for a text editor plug-in the `Task-Content` attribute can be used as follows:

```
[Extension("cap.ui.workbench.taskcontent")]
[TaskContent(FileExtension="txt", Name="Text file")]
public class TextEditorContent : ITaskContent { ... }
```

The `TaskContent` attribute tells the workbench to create a menu item “Text file”. It also notifies the framework about the capability of the plug-in to deal with `.txt` files.

### `cap.ui.workbench.taskcommands`

The "`cap.ui.workbench.taskcommands`" slot is intended to be used for contributing a task command dialog for a particular task content. Task command dialogs are implemented as extensions of .NET's `Form` class. As a task dialog refers to a particular task content window, it is required to identify the task content. This is done with the `TaskCommandFor` attribute. For example, the following code shows a task command extension for the text editor plug-in.

```
[Extension ("cap.ui.workbench.taskcommands")]
[TaskCommandFor("at.dhungana.plugins.texteditor")]
public partial class TextEditorCommands : Form { ... }
```

**`cap.ui.workbench.views`**

In order to add new views to the workbench, clients have to use the extension slot `cap.ui.workbench.views`. The attribute `WorkbenchView` is used to inform the workbench about the availability of a new view. This attribute can be used to specify the name of the view, which is then listed as a menu item in the menu where all other views are listed.

```
[Extension("cap.ui.workbench.views")]
[WorkbenchView("CAP Clipboard")]
public class ClipboardView : Form { ... }
```

### 5.5 Example Plug-ins and Working Environment

To test and demonstrate the platform, a set of plug-ins for a typical rich client workbench have been realized (see Fig. 5 for a sample screenshot) . These are:

- a simple text editor plug-in,
- a web browser plug-in (a public domain implementation has been wrapped and packaged as a CAP.NET plug-in),
- a calendar plug-in,
- a diary plug-in,
- an email plug-in,
- a Sudoku game plug-in,
- and a registry view plug-in for browsing the plug-in registry.

By these plug-in developments it was possible to show that CAP.NET fulfills the requirements of an integration platform for rich client applications and that the user interface paradigm is a simple but appropriate interaction model for typical user tasks.

## 6 Summary and Discussion

In this paper we presented a plug-in architecture and a rich client platform for the .NET platform. Adopting ideas similar to Eclipse, our approach relies on .NET-specific features such as custom attributes, assemblies, metadata and reflection. We argue that the use of these .NET features results in a better plug-in architecture. In particular, we argue that our approach of specifying slots and extensions using custom attributes is more readable and easier to maintain that the Eclipse approach using XML specifications.

In our approach the extension host uses a `Slot` attribute to tag any interfaces that are to be implemented by the contributor. It also uses custom attributes for specifying properties for which the contributor has to provide values. This makes it easy for a contributor to describe an extension. The contributor has to implement the slot's interface by a class and tag this class with the `Extension` attribute. Moreover, the contributor can use the custom attributes defined for the slot to provide values for the required properties. Slots and extensions are specified directly in the source code of an application which makes it easier to keep them in sync with the implementation.

.NET assemblies, as the unit of deployment and versioning, are a most adequate and natural means for the implementation of plug-in components. Furthermore, assemblies can contain arbitrary metadata allowing us to include plug-in specific information. Plug-in discovery is based on .NET metadata and reflection. In .NET 2.0 programmers can load metadata of an assembly without actually loading the code (method `ReflectionOnlyLoad`). This allowed us to realize a lazy loading strategy as in Eclipse, i.e., plug-in integration occurs at start-up time based on metadata without actually loading the code. The code is only loaded when it is activated for the first time.

Hot update means that an old version of a plug-in is unloaded and a new version of it is loaded while the system keeps running. In Eclipse this is accomplished by the OSGi implementation and by the fact that each plug-in is loaded by its own class loader. When the plug-in should be unloaded, the class loader is just disposed and with it the loaded plug-in. .NET works differently in this respect. In .NET, assemblies are loaded into so-called application domains (objects of type `AppDomain`). To unload code one would have to delete the `AppDomain` object. However, application domains also represent memory boundaries and method calls between objects in different application domains have be done using remote method invocation. For that reason, it would represent an unacceptable overhead to load each assembly into its own application domain. Assemblies therefore cannot be unloaded individually. How to realize hot updates in .NET remains an problem that is still to be solved.

## References

1. Beck, K., and Gamma, E.: Contributing to Eclipse. Addison-Wesley, 2003.
2. Cox, B.J.: Planning the software industrial revolution. IEEE Software 7(6), 1990.
3. Crnkovic, I. et al. (eds.): Special Issue: Automated Component-based Software Engineering. Journal of Systems and Software, 74 (1), Elsevier, 2005.
4. Dean, D.: The Security of Static Typing with Dynamic Linking. In Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1997.
5. Dewan, P. and Choudhary, R.: Coupling the user interfaces of a multiuser program. ACM Transactions on Computer Human Interaction, 1995
6. Dhungana, D.: CAP.NET – Client Application Platform in .NET. Master thesis, Johannes Kepler University, Linz, Austria (2006).
7. Eclipse Platform Technical Overview. Object Technology International, Inc., http://www.eclipse.org, February 2003.
8. Ghezzi, C., Monga, M.: Fostering component evolution with C# attributes. International Workshop on Principles of Software Evolution (IWPSE) 2002.
9. Hall, R. S, Cervantes H: An OSGi Implementation and Experience Report; Consumer Communications and Networking Conference, 2004
10. Löwy, J.: Programming .NET Components. O'Reilly Media, Sebastopol (2003)
11. McIllroy, M. D.: Mass produced software components. In: Proceedings of the Nato Software Engineering Conference. 1968, pp. 138-155.
12. Microsoft: Microsoft C# Language Specifications. Microsoft Press, Redmond (2001).

13. mozilla.org: An Introduction To Hacking Mozilla. http://www.mozilla.org/hacking/coding-introduction.html.
14. Mössenböck, H., Beer W., Birngruber, D., Wöß, A.: .NET Application Development. Pearson Addison Wesley, 2004.
15. NetBeans Project: http://www.netbeans.org/index.html
16. OSGi Service Platform, Release 3. The Open Services Gateway Initiative, March 2003, http://www.osgi.org.
17. Shaver, M., and Ang, M.: Inside the Lizard: A Look at the Mozilla Technology and Architecture. http://www.mozilla.org, 2000.
18. Stallman, R.: EMACS: The Extensible, Customizable Display Editor. ACM Conference on Text Processing, 1981.
19. Syperski, C.: Component Software, Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, 2002.
20. Udell, J.: Component Software. BYTE, 19 (5), 1994, pp. 46-55.