

Extending Web Applications with Client and Server Plug-ins¹

Markus Jahn, Reinhard Wolfinger, Hanspeter Mössenböck

Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz
Altenbergerstr. 69, 4040 Linz, Austria
{jahn, wolfinger, moessenboeck}@ase.jku.at

Abstract: Plug-in frameworks support the development of component-based software that is extensible and customizable to the needs of specific users. However, most current frameworks are targeting single-user rich client applications but do not support plug-in-based web applications which can be extended by end users. We show how an existing plug-in framework (Plux.NET) can be enabled to support multi-user plug-in-based web applications which are dynamically extensible by end users through server-side and client-side extensions.

1 Introduction

Although modern software systems tend to become more and more powerful and feature-rich they are still often felt to be incomplete. It will hardly ever be possible to hit all user requirements out of the box, regardless of how big and complex an application is. One solution to this problem are plug-in frameworks that allow developers to build a thin layer of basic functionality that can be extended by plug-in components and thus tailored to the specific needs of individual users. Despite the success of plug-in frameworks so far, current implementations still suffer from several deficiencies:

- (a) *Weak automation.* Host components have to integrate extensions programmatically instead of relying on automatic composition. Furthermore, plug-in frameworks usually have no control over whether, how or when a host looks for extensions.
- (b) *Poor dynamic reconfigurability.* Host components integrate extensions only at startup time whereas dynamic addition and removal of components is either not supported or requires special actions in the host.
- (c) *Separate configuration files.* Composition is controlled by configuration files which are separated from the code of the plug-ins. This causes extra overhead and may lead to inconsistency problems.
- (d) *Limited Web support.* Plug-in frameworks primarily target rich clients or application servers. Although some frameworks extend the plug-in idea to web clients, they are still limited in customization and extensibility: They neither support individual plug-in configurations per user, nor do they integrate plug-ins executed on the client.

¹ This work was supported by the Christian Doppler Research Association and by BMD Systemhaus GmbH.

Over the past few years we developed a client-based plug-in framework called *Plux.NET* which tries to solve the problems described above. While issues (a) to (c) are covered in previous papers [WDP06, WRD08, Wo08, RWG09], this paper deals with issue (d) and describes how Plux.NET can be enabled for multi-user web applications. We show how such web applications can be extended by user-specific plug-ins both on the server side and on the client side. Client-side plug-ins can either run in managed .NET mode [MS08] or in sandbox mode using the Silverlight technology [MS09]. To demonstrate our approach we present a case study, namely a web-based time recorder composed of server-side, client-side, and sandbox components.

Our research was done in cooperation with BMD Systemhaus GmbH, a company offering line-of-business software in the ERP domain. ERP applications consist of many different features that can either be used together or in isolation, thus being an ideal test bed for a plug-in approach.

This paper is organized as follows: Section 2 describes the plug-in framework Plux.NET as the basis of our work. Section 3 uses a case study to explain the architecture of a component-based web application built with Plux.NET. Section 4 compares our work to related research. The paper closes with a summary and a prospect of future work.

2 The Plux.NET framework

Plux.NET [WDPM06, Wo10] is a .NET-based plug-in framework that allows composing applications from plug-in components. It consists of a thin core (140 KBytes) that has slots into which extensions can be plugged. Plugging does not require any programming. The user just drops a plug-in (i.e., a DLL file) into a specific directory, where it will be automatically discovered and plugged into one or several matching slots. Removing a plug-in from the directory will automatically unplug it from the application. Thus adding and removing plug-ins is completely dynamic allowing applications to be reconfigured for different usage scenarios on the fly without restarting the application.

Plug-in components (so-called *extensions*) can have slots and plugs. A *slot* is basically an interface describing some expected functionality and a *plug* belongs to a class implementing this interface and thus providing the functionality. Slots, plugs and extensions are specified declaratively using .NET attributes. Thus the information that is necessary for composition is stored directly in the metadata of interfaces and classes and not in separate XML files as for example in Eclipse [Ec03].

Let's look at an example. Assume that some host component wants to print log messages with time stamps. The logging should be implemented as a separate component that plugs into the host. We first have to define the slot into which the logger can be plugged.

```
[SlotDefinition("Logger")]
[Param("TimeFormat", typeof(string))]
public interface ILogger {
    void Print(string msg);
}
```

The slot definition is a C# interface tagged with a `[SlotDefinition]` attribute specifying the name of the slot ("Logger"). Slots can have parameters defined by `[Param]` at-

tributes. In our case we have one parameter `TimeFormat` of type `string`, which is to be filled by the extension and used by the host. Now, we are going to write an extension that fits into the `Logger` slot:

```
[Extension("ConsoleLogger")]
[Plug("Logger")]
[ParamValue("TimeFormat", "hh:mm:ss")]
public class ConsoleLogger: ILogger {
    public void Print(string msg) {
        Console.WriteLine(msg);
    }
}
```

An extension is a class tagged with an `[Extension]` attribute. It has to implement the interface of the corresponding slot (here `ILogger`). The `[Plug]` attribute defines a plug for this extension that fits into the `Logger` slot. The `[ParamValue]` attribute assigns the value `"hh:mm:ss"` to the parameter `TimeFormat`.

Finally, we implement the host, which is another extension that plugs into the `Startup` slot of the `Plux.NET` core. The extension has a slot `Logger`; this is specified with a `[Slot]` attribute. This attribute also specifies a method `AddLogger` that will be called when an extension for this slot is plugged. `AddLogger` integrates the logger extension and retrieves the `TimeFormat` parameter.

```
[Extension("MyApp")]
[Plug("Startup")]
[Slot("Logger", OnPlugged="AddLogger")]
public class MyApp: IStartup {
    ILogger logger = null; // the logger extension
    string timeFormat; // parameter of the logger extension

    public void Run() {
        ...
        if (logger != null) {
            logger.Print(DateTime.Now.ToString(timeFormat) + ": " + msg);
        }
    }

    public void AddLogger(object s, PlugEventArgs args) {
        logger = (ILogger) args.Extension;
        timeFormat = (string) args.GetParamValue("TimeFormat");
    }
}
```

This is all we have to do. If we compile the interface `ILogger` as well as the classes `ConsoleLogger` and `MyApp` into DLL files and drop them into the plug-in directory everything will fall into place. The `Plux.NET` runtime will discover the extension `MyApp` and plug it into the `Startup` slot of the core. It will also discover the extension `ConsoleLogger` and plug it into the `Logger` slot of `MyApp`. (see Figure 1)

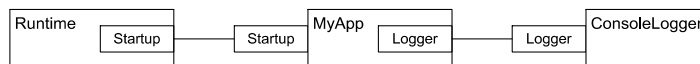


Figure 1: Composition architecture of the logger example

`Plux.NET` offers a light-weight way of building plug-in systems. Plug-ins are just classes tagged with metadata. They are self-contained, i.e., they include all the metadata neces-

sary for discovering them and plugging them together automatically. There is no need for separate XML configuration files. The example also shows that Plux.NET is event-based. Plugging, unplugging and other actions of the runtime core raise events to which the programmer can react. The implementation of Plux.NET follows the plug-in approach itself. For example, the discovery mechanism that monitors the plug-in directory is itself a plug-in and can therefore be replaced by some other way of discovery.

Other features of Plux.NET that cannot be discussed here for the lack of space are *lazy loading* of extensions (in order to keep application startup times small), management of *composition rights* (e.g., which extensions are allowed to open a certain slot, and which extensions are allowed to fill it), as well as a *scripting API* that allows experienced developers to override some of the automatic actions of the runtime core. These features are described in more detail in [Wo10].

3 Extending Plux.NET for the Web

The Internet has become fast and ubiquitous enough for implementing software as web applications that can be accessed by multiple clients. Web applications make it easier to bring software to the market. Additionally, updates can be done centrally without bothering administrators in different companies. However, web applications face similar problems as rich-client applications: when they get too big and feature-rich, they become hard to understand and difficult to use. Furthermore, current web applications are hardly customizable and usually not extensible by users. Finally, it is generally not possible to connect the clients' local hardware to web applications. The goal of our research is to find solutions for these problems.

Our idea is to apply the plug-in approach also to web applications by extending Plux.NET so that it becomes web-enabled. Originally, Plux.NET was designed for single-user rich-client applications. In its extended form it supports multi-user web applications where specific users have their individual set of components and their individual composition state. Users will be able to extend web applications with their own plug-ins or with plug-ins from third party developers.

Plux.NET web components can be classified according to their *composition type* and their *visibility scope*. Depending on their visibility, they can affect just a single user, a group of users (e.g., a department of a company), or all users of a web application. Currently, the composition type of a component can be server-side, client-side, and sandbox integration.

Server-side components are installed and executed on the server. Their advantage is that they are smoothly integrated into the server-based web application. They have minimal communication overhead and maximum availability. However, server-side components increase the work load on the server and constitute a security risk. Users need to be authorized to install their extensions on the server.

Client-side components are placed on the client and plugged into a web application virtually. Their major advantage is that users can integrate their local resources (e.g. hardware) into web applications. Another benefit is that users without authorization for in-

stalling plug-ins on the server can still extend a web application. The disadvantages of client-side extensions are the benefits of server-side extensions: Communication between server-side and client-side components causes a certain overhead. Also, client-side extensions are only available when the client is connected.

A third way of composition is to install components on the server, but to execute them in a client-side sandbox such as Adobe Flash or Silverlight (in the case of .NET). This composition type lends itself for building rich user interfaces in a web browser that go beyond the features of HTML or JavaScript, especially if these interfaces should be extensible and customizable without requiring extensions to be installed on the client. Sandbox extensions are copied to the client on demand and are executed there, so they help to keep the work load on the server small.

Even though components are executed on different computers and in different environments, the development and composition process in Plux.NET is the same for server-side plug-ins, client-side plug-ins, and sandbox plug-ins. The composition model is based on the metaphor of slots and plugs as in the rich client approach. Developers just specify the components' metadata in a declarative way. The runtime core is responsible for plug-in discovery, composition, and communication. Therefore, if there are no dependencies on hardware-specific resources it is possible to reuse a rich client component also as a server-side or a client-side component for web applications. For reusing rich client components as Silverlight sandbox components, they need to be recompiled in a special way, because Silverlight assemblies are not binary compatible with other .NET assemblies.

3.1 Case study and scenarios

To demonstrate the idea of our web-enabled plug-in framework this section shows some scenarios: In a case study we extend a web application by various user-specific plug-ins which are composed by using the different composition types described above and by applying the different visibility scopes to them.

Figure 2 shows the component architecture of a web application which is used for recording and evaluating the labour times of employees. The recording and the statistics are implemented by components, each consisting of a GUI component for rendering the user interface and a component for the business logic. The business logic components (*Recorder*, *Statistics*) are connected to the *Data Provider* component while the GUI components are used by the *Layout Manager* component for building the user interface. The *Layout Manager* is plugged into the root component named *Time Recorder*. All components in Figure 2 are server-side components and thus are installed and executed on the server. To keep the scenario simple, some other required components (e.g., Plux.NET runtime components) are hidden in the following pictures.

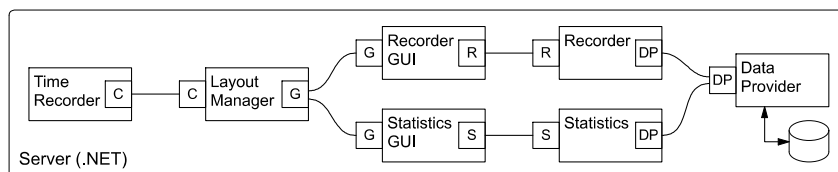


Figure 2: Component architecture of a time recorder web application

To get an impression of how such a web application could look like, Figure 3 shows a possible user interface. The *Layout Manager* arranges the user interfaces of the *Recorder GUI* and *Statistics GUI* components in its window area, depending on some metadata which describes the arrangement of the GUI components. The *Recorder GUI* has buttons for starting, pausing, and stopping time recording while the *Statistics GUI* displays the recorded labour time.

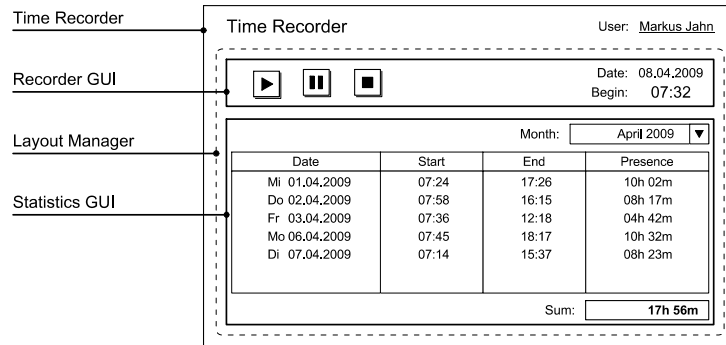


Figure 3: Possible user interface of the time recorder.

Figure 2 contains only server-side components. Thus, the user interface in a client's web browser is restricted to web technologies such as HTML, CSS and JavaScript. However, in our scenario developers want to use the rich internet technology Silverlight for building a more sophisticated user interface. Therefore, they implement the components *Layout Manager*, *Recorder GUI* and *Statistics GUI* as Silverlight components. Since these components are discovered as Silverlight components, they are automatically sent to the client and executed in its Silverlight environment while business logic components stay on the server. Our composition model composes sandbox components in the same way as server-side components. Sandbox components are virtually plugged into server-side components and vice versa. The new composition state is outlined in Figure 4.

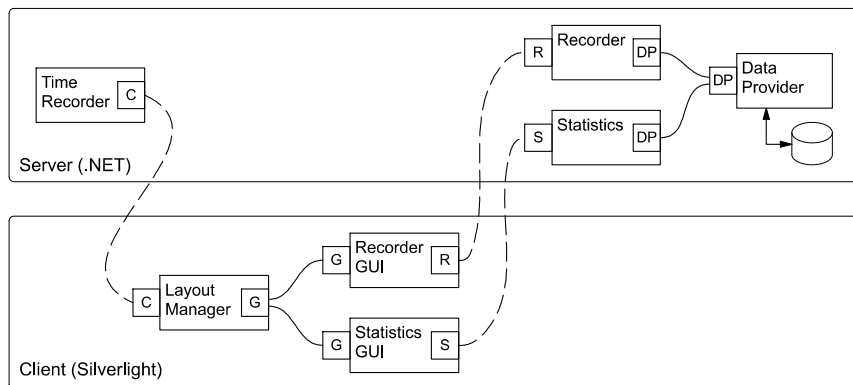


Figure 4: Silverlight components are virtually plugged into server-side components and vice versa. Next, we assume that a user (Client 1) is not fully satisfied with the provided functionality of our time recorder. For annotating his activities during the day he wants to add a

note to each time stamp. Therefore, Client 1 implements his own components for this feature. Since he wants to access his components from any computer, he installs the server-side component *Notes* (a note editor) and the Silverlight component *Notes GUI* on the server. Similar to the other components of our application the server-side component *Note* is used for business logic, while the Silverlight component *Notes GUI* displays the user interface. As Client 1 has set the visibility of these components to private, he is the only user who can see and access them. The individual view of the composition state for Client 1 is shown in Figure 5.

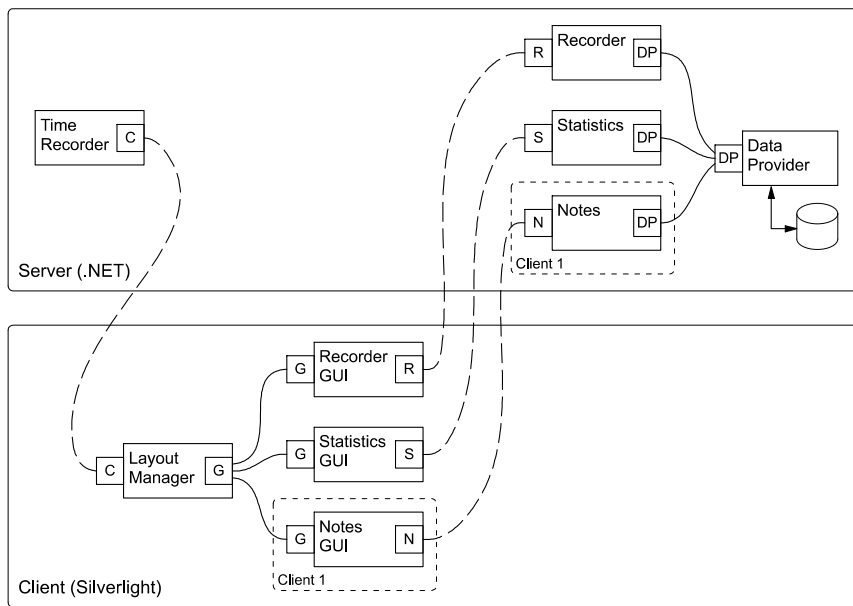


Figure 5: User-specific server-side and sandbox components for Client 1

Besides adding components for a specific user, it is also possible to remove them. For example, a company could deny access to the Silverlight component *Recorder GUI* for several employees. Instead, a hardware time recorder which is represented by a client-side component *Hardware Recorder* could be installed at the company's entrance. The *Hardware Recorder* uses the same server-side *Recorder* component as the Silverlight component *Recorder GUI* did. Figure 6 shows how the Silverlight component *Recorder GUI* is replaced by the client-side component *Hardware Recorder* for clients 2 to 5. Client-side components can be virtually plugged both into server-side components and into Silverlight components and vice versa. If client-side components are connected to Silverlight components the communication between them does not affect the server.

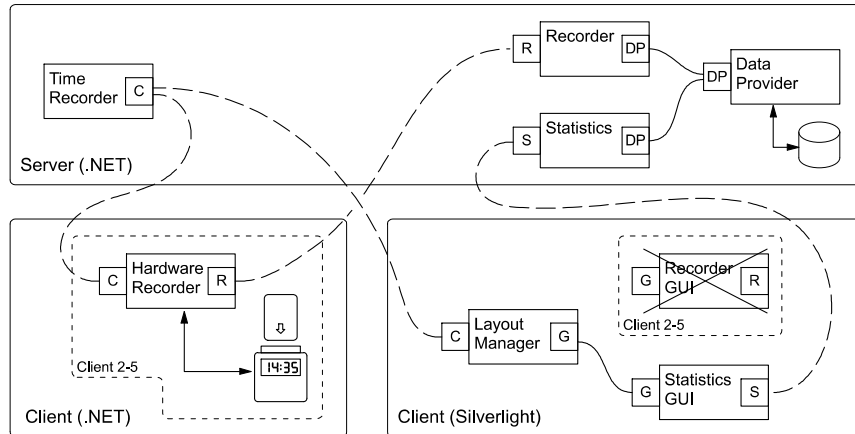


Figure 6: From the view of clients 2 to 5, the Silverlight component *Recorder GUI* is replaced by the client-side component *Hardware Recorder*

3.2 Architecture

We will now look at the architecture and the internals of Plux.NET for web applications. Some aspects described in this section are still under development but the overall architectural design is finished and has been validated with prototypes.

The root component of Plux.NET is the *Runtime* core, which has two slots, one for a discovery component and one for the application's root component. The default discovery component monitors the plug-in directory. Whenever an extension is dropped into this directory its metadata are read and the *Runtime* checks all loaded components for open slots into which the new extension can be plugged. After an extension has been plugged, its own slots are opened and the *Runtime* looks for other extensions (loaded or unloaded) that can fill these slots. These steps are repeated until all matching slots and plugs have been connected.

To perform this composition procedure for multi-user web applications the architecture of Plux.NET had to be extended. We now have different environments in which components can live: There is one environment for server-side components, one for client-side components and one for sandbox components (the latter two exist as separate instances for every client connected to the server). Every environment needs its own runtime infrastructure. So we split the Plux.NET runtime core into a *Server Runtime*, a *Client Runtime* and a *Silverlight Runtime* each running in its own environment. Logically these nodes form a single entity represented by the *Runtime* component on the server (see Figure 7). For discovering extensions on the server and on the client the discovery mechanism also had to be split into a *Server Discovery* and a *Client Discovery* component, each monitoring local extension directories.

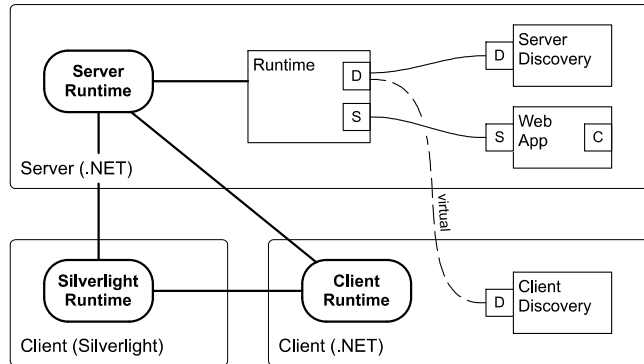


Figure 7: Runtime infrastructure and discovery distributed over several environments

As long as components from the same environment are plugged together everything is like in the rich client case: the local runtime raises a *Plugged* event to which the host component reacts by integrating the extension component. The interesting new feature is *virtual plugging*, which is necessary when the host and the extension live in different environments. If an extension E from environment Env_E should be plugged into a host H from environment Env_H proxies have to be generated on both sides. A proxy H_p representing host H is created in Env_E , and a proxy E_p representing extension E is created in Env_H (Figure 8). These proxies carry the same metadata as the components for which they stand so they can be plugged into matching slots like any other component. If H wants to call a method of E it does the call to the local proxy E_p , which uses the local runtime infrastructure to marshal the call and send a message to proxy H_p in the other environment. H_p then does the actual call to E . Any results are sent back in the same way. Thus the communication between components—from the same or from different environments—is completely transparent to the component developer.



Figure 8: Virtually plugged components and their communication via proxies

Whenever a new extension is discovered on the server or on the client the local discovery component sends a broadcast to the runtime nodes of all other environments. The local runtime nodes decide whether proxies have to be generated. In the case of Silverlight extensions that were discovered on the server the *Silverlight Runtime* on the client requests a transfer of the extension from the server to the sandbox environment on the client.

It is also worth noting that all environments have a consistent copy of the web application's composition state, i.e. they know, which components the application consists of and which plugs are connected to which slots. So the runtime node of every environment can easily find out into which slots a new extension fits and whether the extension has to be plugged in locally or virtually through proxy components.

Since the runtime core is distributed, the individual runtime nodes have to use a communication channel for exchanging the components' metadata and their composition state. Additionally, the runtime core provides the communication infrastructure for the proxies of virtual plugged components. This infrastructure is based on the Windows Communication Foundation (WCF) API [Ca07]. As WCF supports several communication standards for distributed computing, the actually used communication technology can be configured by administrators. In our case study we used SOAP [GHM07] in combination with HTTP.

Since web applications can be used by many clients simultaneously, the runtime core has to deal with several composition states in parallel. In doing so it has to make sure not to run out of memory. Hence, it applies a well-known solution for this problem: Once the server has only little memory left, server-side components get released at the end of a request and are recreated for new requests. This approach ensures that web applications can scale up to serve many simultaneous requests without running out of server memory. This concept also enables the usage of server farms. The drawback is that composition and component states need to be persisted during successive requests. The composition state is persisted automatically while the component state has to be persisted by the components themselves using the infrastructure of the runtime core. Components can either use custom .NET attributes to declare which values should be persisted and restored, or they can react to automatically raised events for persisting and restoring.

The runtime does not only persist the server-side state, but also the states of the client environments. Thus, no matter from which computer a client connects to the application, it will always get the same state as it had last time, provided that possibly used client-side components are available on each computer.

4. Related work

Plug-in frameworks have become quite popular recently. However, most of them are either targeting rich-client applications, or have no dynamic composition support, or aim at web applications, but cannot be customized and extended by end users.

Eclipse [Ec03] is probably the most prominent plug-in platform today. It is written in Java and since version 3.0 it is based on the OSGi framework *Equinox* [Eq09]. Like Plux.NET it consists of a thin core and a set of plug-ins that provide further functionality. The major differences between Eclipse and Plux.NET are the following: Eclipse declares the metadata of plug-ins in XML files while Plux.NET declares them directly in the code using .NET attributes. Eclipse supports only rich client applications while our approach targets also multi-client web applications with extensions both on the server and on the client. Most importantly, the composition of Eclipse plug-ins has to be done manually, i.e. the host component has to use API calls to discover plug-ins, read their metadata and integrate them. In Plux.NET, plug-ins are discovered automatically and all matching slots are immediately notified by the runtime core, thus automating a substantial amount of composition work. Although Eclipse allows plug-ins to be added dynamically, the code for integrating plug-ins at startup time and at run time is different, whereas Plux.NET uses the same uniform mechanism for both cases.

Many component-based web applications are based on the *Java Enterprise Edition* (Java EE) [Su09]. Java EE is a software architecture that allows the development of distributed, component-based, multi-tier software running on an application server. Java EE applications are generally considered to be three-tiered where components can be installed on the Java EE server machine, the database machines, or the client machines. Web components are usually servlets or dynamic web pages (created with JavaServer Faces or Java Server Pages) running on the server, but they can also be defined as application clients or applets running on the client. Even though Java EE provides a framework for building component-based and distributed web applications it provides no automatic composition support for components. Composition has to be done programmatically. Moreover, Java EE does not provide an individual composition state for every end user, so users cannot customize or extend web applications for their special needs.

A further component system for distributed, component-based applications is *SOFA 2* [HP06, BHP06, BHP07]. It implements a hierarchical component model with primitive and composite components. Primitive components consist of plain code whereas composite components consist of other subcomponents. SOFA 2 has a distributed runtime environment which automatically generates connectors to support a transparent distribution of applications to different runtime environments. For communication the connectors use method invocation, message passing, streaming, or distributed shared memory. SOFA 2 allows dynamic reconfiguration of applications by adding and removing components as well as dynamic update of components at run time. In contrast to Plux.NET, however, SOFA 2 needs an ADL (Architecture Description Language) for describing the composition of components. Plux.NET does not need such a specification; its runtime composes an application on the fly using the declarations of slots and plugs provided by the Plux.NET components. We argue that this concept is more flexible and easier to maintain than a global ADL specification. Finally, browser-based web applications are not supported by SOFA 2. It has no multi-user support for individual composition states and no mechanism for persisting and restoring the composition states at run time.

Currently, the only way how end users can extend a web application is through client-based plug-in systems such as *Mozilla Firefox* [Mo09]. However, client plug-ins are not integrated into web applications. They only add usability features to user interfaces or enable web browsers to use advanced web technologies such as Flash, Silverlight, or Java Applets.

5. Summary and future work

In this paper we presented a dynamic plug-in framework for rich client and web applications with the focus on multi-user web applications that are extensible by end users. Each client can install its individual set of components and has its personal composition state. Components can be instantiated in different environments and on different computers, but are transparently composed into a single web application. Components for business logic can stay on the server, components that are connected to a client's local resources can be executed on the client-side, and user interface components can live in a client's rich internet environment such as Silverlight.

Since several aspects mentioned above have been realized only prototypically so far, we are still improving the distributed runtime core of Plux.NET. Furthermore, we are working on a security model for plug-ins, a layout manager for extensible component-based user interfaces, a keyboard shortcut manager and many other helpful tools for component-based software development.

References

- [BHP06] Bures, T., Hnetyuka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, Seattle, USA, IEEE CS, ISBN 0-7695-2656-X, pp.40-48, August, 2006.
- [BHP07] Bures, T., Hnetyuka, P., Plasil, F., Klesnil, J., Kmoch, O., Kohan, T., Kotrc, P.: Runtime Support for Advanced Component Concepts, Proceedings of SERA 2007, Busan, Korea, IEEE CS, ISBN 0-7695-2867-8, pp. 337-345, August, 2007.
- [Ca07] Chappell, D. (Microsoft): Introducing Windows Communication Foundation, <http://msdn.microsoft.com/en-us/library/dd943056.aspx>, September, 2007.
- [Ec03] Eclipse Platform Technical Overview. Object Technology International, Inc., <http://www.eclipse.org>, February 2003.
- [Eq09] Equinox Mission Statement., <http://www.eclipse.org/equinox/>, 2009.
- [GHM07] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Karmarkar, A., Lafon, Y.: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C Recommendation, April, 2007.
- [HP06] Hnetyuka, P., Plasil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models, Proceedings of CBSE 2006, Vasteras near Stockholm, Sweden, LNCS 4063, ISBN 3-540-35628-2, ISSN 0302-9743, pp. 352 - 359, (C) Springer-Verlag, June, 2006.
- [Mo09] Mozilla Foundation: Firefox Browser, Free ways to customize your Internet, <http://www.mozilla.com/en-US/firefox/personal.html>, 2009
- [MS08] Microsoft .NET Framework. <http://www.microsoft.com/net/overview.aspx>, 2008.
- [MS09] Microsoft Silverlight, <http://silverlight.net/>, 2009.
- [RWG09] Rabiser, R., Wolfinger, R., Grünbacher, P.: Three-level Customization of Software Products Using a Product Line Approach. 42nd Hawaii International Conference on System Sciences, HICSS-42, Big Island, Hawaii, USA, January, 5-8, 2009.
- [Su09] Sun Microsystems, Inc.: The Java EE 6 Tutorial, Volume I, Basic Concepts Beta, <http://java.sun.com/javae/6/docs/tutorial/doc/JavaEETutorial.pdf>, August, 2009.
- [WDP06] Wolfinger, R., Dhungana, D., Prähofer, H., Mössenböck, H.: A Component Plug-in Architecture for the .NET Platform. Modular Programming Languages, Lightfoot, David; Szyperski, Clemens (Eds.), Lecture Notes in Computer Science , Vol. 4228, Proceedings of 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006.
- [WRD08] Wolfinger, R., Reiter, S., Dhungana, D., Grünbacher, P., and Prähofer, H.: Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. 7th IEEE International Conference on Composition-Based Software Systems, ICCBSS 2008, Madrid, Spain, February, 25-29, 2008.
- [Wo08] Wolfinger, R.: Plug-in Architecture and Design Guidelines for Customizable Enterprise Applications, OOPSLA 2008 Doctoral Symposium, OOPSLA 2008, Nashville, Tennessee, October, 19-23, 2008.
- [Wo10] Wolfinger, R.: Dynamic Application Composition with Plux.NET: Composition Model, Composition Infrastructure. PhD Thesis, Johannes Kepler University, Linz, Austria, 2009.