JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF
Faculty of Engineering
and Natural Sciences

Plugin-based Distributed Multi-user Web Applications with Plux

DISSERTATION

submitted in partial fulfillment of the requirements
for the academic degree

Doktor der Technischen Wissenschaften

Submitted by

DI Markus Jahn

At the

Institute for Systemsoftware

Accepted on the recommendation of

O. Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Doc. RNDr. Tomáš Bureš, Ph.D.

Co-advisor

Dr. Reinhard Wolfinger

Linz, July 2014

Sworn Declaration

I hereby declare under oath that the submitted doctoral dissertation has been written solely by me without any outside assistance, information other than provided sources or aids have not been used and those used have been fully documented.

The dissertation here present is identical to the electronically transmitted text document.

Linz, July 2014 Markus Jahn

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, July 2014 Markus Jahn

Abstract

Despite the fact that off-the-shelf software applications tend to become more and more feature-rich, they are still often felt to be incomplete, because it will hardly ever be possible to hit all requirements of a user out of the box, regardless of how big and complex an application is. For desktop applications, plugin frameworks are a solution for this problem, which allows developers to build a thin layer of basic functionality that can be extended by plugin components and thus tailored to the needs of individual users. For web applications on the other hand, existing plugin frameworks are not suitable to enable users to install their user-specific plugins. However, as web applications increasingly supersede desktop applications, web applications also become feature-rich and therefore should also be extensible and customizable in order to tailor it to the needs of individual users.

A number of web platforms allow developers to componentize web applications. However, as web applications are executed on a web server, but not on each client-side computer individually, in existing solutions only developers can benefit from this modularity, whereas users cannot adapt web applications with components. Moreover, existing solutions only allow changing the set of components for a web application, but they do not make web applications user-customizable, because all users are using the same set of server-side components. Developers can make their programs somewhat user-customizable, e.g., by allowing the users to adjust the user interface or to disable specific features.

This thesis presents Plux for Web, a component model and a component infrastructure for building plugin-based web applications that are customizable and extensible with individual components per user and that can be distributed among multiple computers. Thus, user-specific plugins can either be installed on the web server, or they can be installed on the users' client-side computers.

The component model defines a metadata standard that allows adding and removing plugins in a plug-and-play manner, a deployment standard that maintains local and remote plugins for individual users, a composition standard that connects independent plugin components seamlessly to a coherent web application, an interaction standard that enables local and distributed communication between plugin components, and a customization standard that maintains optional settings for plugins. The component infrastructure implements the component model, and thus provides a platform that can assemble plugin-based distributed user-specific web applications.

Kurzfassung

Obwohl Standardsoftware immer mehr Funktionalität bietet, vermissen Anwender trotzdem oft Funktionen für ihre ganz individuellen Anforderungen. Es ist nahezu unmöglich, dass Standardsoftware in ihrem Auslieferungszustand alle Anforderungen unterschiedlicher Anwender erfüllen. Für Desktopanwendungen bieten Plug-in-Frameworks eine Lösung für dieses Problem. Diese erlauben Softwareentwicklern kompakte Kernanwendungen mit vielen Grundfunktionen zu entwickeln, die später von den Anwendern durch individuelle Plug-ins erweitert werden können. Für Webanwendungen bieten existierende Plug-in-Frameworks allerdings keine Lösung, die es Anwendern ermöglicht, ihre individuellen Plug-ins zu installieren. Da Webanwendungen an Bedeutung gewonnen haben, sollten auch sie individuell erweitert werden können, um sie an unterschiedliche Anforderungen anpassen zu können.

Einige Webplattformen unterstützen Softwareentwickler bei der Erstellung von komponentenbasierten Webanwendungen. Da Webanwendungen allerdings auf einem Webserver ausgeführt werden und nicht auf den jeweiligen Computern der Anwender, können nur Softwareentwickler oder Systemadministratoren die Modularität einer solchen komponentenbasierten Webanwendung nutzen. Anwender können solche Webanwendungen nicht mit ihren eigenen Komponenten anpassen. Außerdem ermöglichen existierende Techniken nur das Austauschen von Komponenten für alle Benutzer einer Webanwendung, nicht aber individuell für verschiedene Anwender. Üblicherweise können Softwareentwickler Webanwendungen nur für Anwender anpassbar machen, indem sie erlauben das Aussehen der Benutzeroberfläche zu verändern oder verschiedene Funktionen ein- und auszuschalten.

Diese Dissertation präsentiert das Komponentenmodell Plux für Webanwendungen und eine Kompositionsinfrastruktur, die dieses Komponentenmodell implementiert. Plux ermöglicht Anwendern ihre Webanwendungen mit individuellen Plug-ins anzupassen und zu erweitern. Dabei können Komponenten verteilt auf unterschiedlichen Computern ausgeführt werden, wodurch Plug-ins entweder am Webserver oder auf den Anwender-Computern installiert werden können.

Das Komponentenmodell spezifiziert: einen Metadaten-Standard, der es erlaubt Plug-ins per Plug-and-play zu einer Webanwendung hinzuzufügen oder zu entfernen; einen Deployment-Standard, der lokale und verteilte Plug-ins für verschiedene Anwender individuell verwaltet; einen Kompositions-Standard, der voneinander unabhängige Plug-ins nahtlos zu einer einheitlichen Webanwendung verbindet; einen Interaktions-Standard, der lokale und verteilte Kommunikation zwischen Plug-ins ermöglicht; und einen Konfigurations-Standard zur Verwaltung von optionalen Konfigurationsdaten für Plug-ins. Die Kompositionsinfrastruktur implementiert das Komponentenmodell und bietet eine Plattform zur Entwicklung von plug-in-basierten, verteilten und mehrbenutzerfähigen Webanwendungen.

Table of Contents

Introduction

Plugin architectures are well suited for building extensible and customizable applications from components. Users of web applications would also benefit from plugin architectures, because they could even tailor their web applications with individual plugins as they are used to in component-based desktop applications. As current plugin systems only target desktop applications, a solution for web applications that are extensible and customizable by users is needed. This chapter presents the problems of current plugin systems. It introduces the research project that constitutes the context for this thesis, gives an overview of the research contributions, and outlines the remaining chapters of this thesis.

Despite the fact that software applications tend to become more and more feature-rich, they are still often felt to be incomplete, because it will hardly ever be possible to hit all requirements of a user out of the box, regardless of how big and complex an application is. For desktop applications, plugin frameworks are a solution for this problem, which allows developers to build a thin layer of basic functionality that can be extended by plugin components and thus tailored to the needs of individual users. However, as web applications increasingly supersede desktop applications, web applications also become feature-rich and therefore should also be extensible and customizable to tailor to the needs of individual users.

A number of web platforms allow developers to componentize web applications. However, in existing solutions only developers can benefit from this modularity, whereas users cannot adapt web applications with components. Moreover, existing solutions only allow changing the set of components for a web application, but they do not make web applications user-customizable, because all users are using the same set of components. Developers can make their programs somewhat user-customizable, e.g., by allowing the users to adjust the user interface or to disable specific features. However, this must be individually programmed into each web application and is not supported by the web platform.

Another issue that needs to be individually programmed is access to client-side resources (e.g., integration of client-side software or hardware such as a point-of-sale

terminal), because the integration of user-specific client-side plugins into server-side web applications is also not supported by existing plugin systems.

This thesis presents Plux for Web, a component model and a component infrastructure for building plugin-based web applications that are customizable and extensible with individual components per user and that can be distributed across multiple computers. The component model defines the metadata standard, the deployment standard, the composition standard, the interaction standard, and the customization standard for distributed web applications. The component infrastructure implements the component model, and thus provides a platform that can assemble user-specific web applications by seamlessly integrating user-specific components, which can be installed on server-side and on client-side computers.

1.1 Research Context

This thesis was realized as part of the industrial research project: *Component architectures for next-generation business computing systems*. One of this project's goals is to design and implement a component model and a composition infrastructure for extensible and customizable web-based enterprise applications. The project is funded by and conducted in close cooperation with *BMD Systemhaus GmbH* and the *Christian Doppler Laboratory for Automated Software Engineering* associated with the *Institute of System Software* at the *Johannes Kepler University Linz*. BMD builds enterprise resource planning software for small and medium-sized companies in Austria, Germany, and Hungary.

Earlier in this project, the Plux component model and composition infrastructure for desktop applications [Wolfinger, 2010] was developed as the basis for the next-generation business applications of BMD. Programs developed with Plux are built with fine-grained components, which are assembled by Plux, using a plug-and-play approach. Users can adapt the program on-the-fly to the working situation at hand by adding, removing, or swapping the set of components to be used.

BMD offers its enterprise software as a desktop application as well as a browser-based web application. They want to offer the flexibility provided by Plux both for desktop applications and for their web application. The difference between the desktop application and the web application is that the latter must support multiple users and component distribution over the network. Multi-user support means that the enterprise application should be customizable for an individual user to meet his specific needs without interfering with the other users. Component distribution support means that the enterprise application should be seamlessly assembled from plugin components that can reside on the web server as well as on the client-side computer.

The next section discusses the problems that must be solved to support multiple users and component distribution in dynamically composed web applications.

1.2 Problem Statement

Despite the success of plugin systems for desktop applications, they still suffer from several deficiencies in the domain of web applications. This thesis discusses the problem of how to assemble web applications such that every user can have his individual set of components and how to allow these components to reside on different computers. Existing web frameworks and plugin systems do not provide a solution for this problem, as they lack support for the following issues to be solved. How these issues can be solved is an open research problem.

- *User-specific plugins.* In web applications that are built with existing web frameworks and plugin systems, applications are assembled with the same set of components for every user. Such web applications do not allow individual users to tailor the program to their needs, e.g., by integrating their user-specific components. Of course, the programmer of a web application could program such a feature manually, but this effort must be repeated for every web application. A reusable solution for user-specific plugins is missing.

- *Dynamic reconfiguration.* Existing web frameworks do not support the adaptation of a web application without restarting it. Dynamic reconfiguration is particularly useful in combination with user-specific plugins, because it would be impractical to restart a web application every time a user adds plugins.

- *Distributed plugins.* Web applications that are built with existing plugin technology typically require plugins to reside on the web server. Such applications cannot integrate user-specific plugins from the client computer; this would be necessary, for example, in order to connect client-side hardware like a barcode scanner to the web application. A solution for distributing components of a web application to multiple computers is not available.

- *Automatic composition*. Web frameworks typically do not connect components of a web application automatically. Even if web frameworks are combined with plugin systems that support automatic composition, distributed components still need to be connected programmatically. Support for automatic composition of distributed components is not available. Automatic composition for local and for remote components would be useful to enable users to change user-specific components on the server side and on the client side without causing any extra programming effort for the developers of a web application.

- *Implementation transparency for distributed components.* Plugin systems that support distributed components use technologies such as remoting or web services for communication between these components. As such technologies do not support distributed thread management, reference identity, and data

synchronization for serialized objects, distributed components must be implemented in a different way than local components. What we would like to have is implementation transparency, i.e., we would like to implement each component in the same way, regardless of whether it is connected locally or remotely. Implementation transparency would also allow users to install the same component either on the server side or on the client side, as their implementations do not differ.

1.3 Research Contributions

This thesis claims the following research contributions, which are combined into a single coherent component model for building plugin-based web applications that are customizable and extensible with individual plugins per user and that supports plugin distribution across multiple computers:

- *Automatic composition.* The thesis presents an automatic composition approach that allows adding and removing plugins in a plug-and-play manner regardless if plugins are deployed locally on a single computer or if they are distributed across multiple computers. The composition infrastructure uses the self-contained metadata of plugins to retrieve requirements and provisions of components and connects them automatically.

- *Declarative composition behaviors*. The thesis presents composition behaviors, which are reusable composition logic that can be attached declaratively to components. Composition behaviors can influence the automatic composition process, e.g., to ensure a certain composition order for components, which may are dependent on the existence of other components; to cancel certain composition operations because of unfulfilled preconditions, or react on composition events to trigger new composition operations. Composition behaviors encapsulate such composition logic into a reusable composition library.

- *User-specific plugins*. This thesis presents an approach that enables users to extend and customize web applications with their individual user-specific plugins. The same application can be extended in different ways by different users at the same time without affecting other users. For this, the composition infrastructure maintains individual composition states for users in separated user scopes.

- *Distributed plugins*. This thesis presents an approach for distributed plugin components that enables users to install plugins either on the server-side or on the users' client-side computers. The distribution of plugins is transparent to plugin developers, i.e., the implementation of plugins remains the same

regardless if they are connected locally on the same computer or remotely on different computers. For this, the presented component model specifies distributed thread management, reference identity and data synchronization for distributed objects, as well as a distributed garbage collection mechanism.

- *Composition infrastructure.* The thesis presents the design and implementation of a composition infrastructure that implements the component model that is specified in the thesis. The infrastructure provides a platform that assembles user-specific web applications from plugin components that can be distributed across multiple computers. It composes web applications dynamically, i.e., the web application can be reconfigured by swapping sets of components without restarting the application.

1.4 Project History

Plux for Web is part of the Plux project, which researches composition infrastructures for dynamically composed applications. The original Plux project [Wolfinger, 2010] targeted desktop applications only. The Plux research project is conducted by the Christian Doppler Laboratory for Automated Software Engineering associated with the Institute for System Software at the Johannes Kepler University Linz, in cooperation with the industry partner BMD Systemhaus GmbH.

At the time of this writing the Plux team comprises: the project manager Reinhard Wolfinger; Markus Löberbauer, who works on testing and debugging of dynamically composed applications; and the Ph.D. student Markus Jahn, who works on dynamically composed web applications.

Our industry partner BMD Systemhaus initiated the project in an effort to build the basis for their next-generation enterprise application, which comes in a desktop and a web variant. Both variants should be extensible with third-party plugins and reconfigurable at run time.

In 2006 we designed a component model based on the metaphor of slots and extensions [Wolfinger et al., 2006]. In 2007, we published a composition infrastructure and demonstrated novel applications, which can be reconfigured in a plug-and-play manner. For the first time, users could add components to a program and remove components from a program to adapt it to their working situation at hand, without programming, configuring, or even restarting the program. A further novelty was a visualizer that instantly shows the program's architecture and its changes. Furthermore, we published integration models for secure integration of untrusted third-party plugins [Wolfinger and Prähofer, 2007].

From 2008 to 2010 we redesigned Plux to reduce the programming effort for component developers. This has been accomplished with a richer composition model [Jahn et al., 2011], composition behaviors [Jahn et al., 2010a], and component templates

[Wolfinger et al., 2010]. With the richer composition model, components can share information in a standardized manner; with behaviors, composition logic can be reused to control complex composition scenarios declaratively; and with component templates, custom components can be generated from generic component templates.

From 2010 to 2012 Plux was extended to support distributed multi-user web applications [Jahn et al., 2010b; Jahn et al., 2011]. We designed a method for composability testing and composition debugging [Löberbauer et al., 2010; Löberbauer et al., 2012] and implemented a composability test tool and a composition debugger [Lengauer, 2012]. We also created a model for retrofitting security in component-based programs [Wolfinger et al., 2012] and a security manager for license enforcement and retrofitted security in Plux [Hribernig, 2012].

During the whole project the following student projects were conducted based on Plux: Stephan Reiter and Christian Mittermair componentized a customer relationship management application [Reiter and Wolfinger, 2007; Mittermair, 2010], Markus Jahn created a cross compiler infrastructure [Jahn, 2009], Mario Eder created a web site monitor [Eder, 2008], Rainer Pichler created a tool to record run-time statistics [Pichler, 2009], Zoltan Toth created a script interpreter for composition scripts, Andreas Gruber created a graphical composition tool for Plux programs [Gruber, 2010], Sabine Weiss created a highly extensible customer relationship management application [Weiss, 2010], Patrick Hagmüller ported the core elements of Plux from C# to Delphi [Hagmüller, 2013], Bernhard Schenkermayr created a highly customizable calculator [Schenkermayr, 2013], Thomas Reinthaler created an application builder [Reinthaler, 2012].

1.5 Structure of the Thesis

Chapter 2 discusses the state of the art for component systems from three angles: it looks at systems for local components, at systems for distributed components, and at systems for web application components. It defines the component terminology and the distributed systems terminology used in this thesis.

Chapter 3 describes the Plux component model, which specifies metadata, deployment, composition, interaction, and customization standards for Plux components.

Chapter 4 describes the benefits of the Plux component model for the web with typical scenarios for component-based distributed multi-user web applications that are dynamically reconfigurable. It uses a case study of a web application for recoding working hours.

Chapter 5 describes the Plux component model for the web, which extends the component model from Chapter 3 with specifications of deployment, composition and interaction standards required for distribution, multi-user, and web support.

Chapter 6 describes the Plux composition infrastructure, which implements the Plux component model and allows executing distributed multi-user web applications built from Plux components.

Chapter 7 evaluates the approach of building component-based distributed multi-user web applications with Plux by the use of case studies. It presents a number of statistics concerning the degree of component reusability, memory consumption and delay time issues by comparing component-based desktop applications with component-based distributed multi-user web applications.

Chapter 8 summarizes the contributions of this thesis, addresses open issues of the current approach, presents ideas for future research, and concludes with an overview of the current state.

State of the Art

The idea of composing desktop applications from loosely coupled, prefabricated, and reusable software components is already pursued since several decades. More recently it became popular to use plugin frameworks in order to make desktop applications customizable and extensible with third-party plugins. However, in times of the internet, desktop applications are increasingly overtaken by web applications. There are many web application frameworks that support developers in building web applications. Although such frameworks embrace component technology, they have not yet picked up the concept of plugin components. The idea of Plux for web applications is to bring both worlds together, plugin component systems and web application frameworks. This chapter overviews the history of both worlds from three angles, namely components for applications that run on a single computer, such that are distributed across multiple computers, and such that are used specifically to build web applications. Furthermore, it defines the component terminology used in this thesis.

In contrast to monolithic applications, component-based applications enable developers to reuse program logic implemented by other developers, as well as to replace an implementation of one manufacturer with an implementation of another, without affecting the rest of the application. Furthermore, using composition, developers can assemble programs from components without programming effort. With plugin components, end users also can extend and customize their applications, e.g., by adding features for their specific needs and by removing features they do not need. Distributed component systems enable developers to build applications that are distributed across multiple computers in different locations, e.g., to connect the applications of businesses that work together. Web application frameworks enable developers to build applications where the application runs on a web server and the users access it using a web browser, thus minimizing the deployment effort for the manufacturers and the maintenance effort for the users.

This chapter presents the history, the terminology, and the capabilities of technologies that are relevant for this thesis. These technologies are grouped into component technologies, distribution technologies, and web technologies in the following sections:

Section 2.1 gives a historical overview from components to component-based web technologies. Section 2.2 defines the component terminology, the distribution terminology, and the web terminology. Section 2.3 describes capabilities that are required to build component-based distributed web applications, evaluates which of these capabilities are supported by existing technologies and concludes this chapter with an overview of deficiencies of current technologies.

2.1 Historical Overview

Since this thesis deals with plugin-based distributed web applications, this section deals with a historical perspective of the underlying technologies, namely components, distribution, and web technology.

2.1.1 Component Technologies

The idea of component systems goes back to [McIlroy, 1968] who devised that software should be componentized, i.e., built from prefabricated components, in order to overcome the so-called software crisis. The term software crisis described the problems that resulted from rising complexity of computer programs. The first component systems that are still relevant today appeared 1964, as *Dynamic Link Libraries* (today dll files in Windows) or *Dynamic Shared Objects* (Unix SO). Dynamic linking refers to components that are linked while a program is loaded and was originally developed in the *Multics* operating system [Schell, 1971].

The following sections present the history of today's component technology structured by the major platforms *Microsoft*, *Object Management Group*, and *Java*, as well as *Web Component Technologies* and *Academic Component Technologies*.

Microsoft Component Technologies

Microsoft started developing the *Component Object Model (COM)* [Microsoft, 2012c and Box, 1998] in 1987. COM emerged from *Dynamic Data Exchange (DDE)* [Microsoft, 2012a], a technology used to implement the clipboard in the *Windows* operating system. In 1992, DDE evolved into *Object Linking and Embedding (OLE)* [Microsoft, 2012b], a service to embed *Microsoft Office* documents into other documents. OLE allowed, for example, embedding a spreadsheet into a text document and editing the spreadsheet from within the word processor without switching applications. In 1995, the COM specification was published and it is still used in many services of the Windows operating system, e.g., the *Windows Explorer* can be extended with COM components. In 1996, COM was extended into the *Distributed Component Object Model (DCOM)* [Microsoft, 2012d]. DCOM supports data exchange beyond computer boundaries for distributed applications, using *Microsoft Remote Procedure Calls (MSRPC)* [Microsoft, 2003]. Facing the complexities of OLE and the poor support for it in development tools, Microsoft simplified the specification in 1996 and rebranded the

technology as *ActiveX* [Microsoft, 1996 and The Open Group, 1999]. ActiveX was used in the *Internet Explorer* web browser in order to embed active content into web pages. Such ActiveX components are installed on the server, automatically downloaded to the client, and executed in the browser. The development culminates in 1999 into *COM+* [Microsoft, 2012e], which comprises DCOM and additional services, such as *Microsoft Transaction Server (MTS)* [Microsoft, 1998] for distributed transactions, and as well as *Microsoft Message Queuing (MSMQ)* [Microsoft, 2012f] for asynchronous inter-application messaging. Today, COM+ is still an essential part of the Windows operation system.

In 2002, Microsoft presented the *.NET framework* [Microsoft, 2012g], as a successor of COM+. In .NET components are deployed as assembly files, which contain IL code, metadata, and resources. .NET is language-independent, i.e., programs can be written in multiple programming languages, such as C# and Visual Basic. .NET includes a large base class library with support for remoting, web services, and web applications.

In 2010, Microsoft released the *Managed Extensibility Framework (MEF)* [Microsoft, 2010]. MEF is a plugin framework that is based on .NET and uses .NET attributes to specify component metadata, which are used by the MEF composition engine to connect components with their required services. Currently MEF is used in the *Microsoft Visual Studio IDE* [Microsoft, 2012h] to make it extensible and customizable via third-party plugins.

OMG Component Technology

In 1991, the Object Management Group, an international consortium of renowned corporations, specified the *Common Object Request Broker Architecture (CORBA)* [OMG, 2012]. CORBA is a middleware standard for distributed components that allows interacting software components to be distributed across multiple computers. CORBA is a binary standard, which is implemented in various programming languages, available for multiple platforms, and supports multiple communication protocols. Since 1991 the development of CORBA is ongoing: the specification of CORBA 2.0 was released in 1997, the specification of CORBA 3.0 was released in 2002, and the current specification is CORBA 3.3, which was released in November 2012.

Java Component Technologies

In 1996, Sun Microsystems developed a component system called *Java Beans* [Sun Microsystems, 1996]. Beans basically are plain Java objects conforming to the Java Beans convention, i.e., a bean implements a defined set of properties as getter and setter methods. A bean can encapsulate multiple Java objects, can be serialized, and thus can be transported over the network. Developers can use, for example, the bean workbench to compose an application from beans, serialize composed applications, and deploy them to users, without any programming effort [Beer, 2000].

In 1996 Jaroslav Tulach started together with a group of students the NetBeans project [Oracle, 2013b] at the Faculty of Mathematics and Physics of the Charles University in Prague. NetBeans is an application framework, which allows developers to compose applications from a set of modules. Initially NetBeans was called Xelfi, because the goal of the project was to write a Java IDE similar to the Delphi IDE. The name NetBeans was derived from network JavaBeans. The NetBeans team wanted to create an abstraction of the network and enable developers to manipulate network JavaBeans from any IDE. However, extending the existing IDEs with plugins was very difficult. Therefore the NetBeans team decided to use the experience from Xelfi to create a new modular IDE. In 1999 NetBeans DeveloperX2 was released, which was redesigned for a modular architecture and which forms the basis of the current NetBeans platform. Furthermore, also in 1999, Sun Microsystems became involved in the NetBeans project. A few months later, NetBeans became an open-source project. Many developers started using the NetBeans IDE's platform to build their own applications from their own plugins. In 2010 Oracle acquired Sun Microsystems and NetBeans became a part of Oracle, which continues work on the NetBeans project. [Oracle, 2013c; Oracle, 2013d]

In 1997, IBM released *Enterprise Java Beans (EJB)* [Vatkina, 2013], a server-side component architecture for modular enterprise applications. EJB is used to implement application servers and targets concerns such as persistence, transactions, security, and distribution. EJB uses *Java Remote Method Invocation (RMI)* [Oracle, 2010], an API for distributed communication based on an object-oriented version of *Remote Procedure Calls* [Thurlow, 2009].

In 2000, the *Open Services Gateway Initiative Alliance (OSGi)* [OSGi Alliance, 2012a] published a specification for the OSGi component system. The Java-based OSGi provides a service registry that is used by components to retrieve their requested services. OSGi implementations are available from multiple vendors, e.g., *Knopflerfisch* [Makewave, 2013], *Apache Felix* [The Apache Software Foundation, 2008], or *Eclipse Equinox* [Equinox, 2012].

In 2001, IBM presented *Eclipse* [Eclipse, 2006], an integrated development environment (IDE) for Java. Eclipse is plugin-based so that users can assemble a custom IDE for their specific needs from plugin components. Since 2004 Eclipse is based on the OSGi implementation Equinox [Equinox, 2012]. Eclipse evolved into the most outstanding representative for plugin infrastructures, because various third-party developers provide a vast number of plugins, e.g., for version control, bug tracking, or profiling.

In 2007, the *Eclipse Rich AJAX Platform (RAP)* [RAP, 2012; Muskalla and Sternberg, 2007] was introduced. Since Version 2.0, RAP was renamed to *Remote Application Platform*. It allows building web applications using Eclipse plugins. Thereby a server-side Equinox environment loads the Eclipse plugins onto a web server and RAP renders the user interface for web browsers using technologies such as *HTML* [Berjon et al., 2013], *JavaScript* [Ecma, 2011], and *JavaScript Object Notation (JSON)* [Crockford, 2006].

Netscape Component Technology

The most frequent use of plugin components in the web are browser plugins. Browser plugins enable users to customize web browsers, which are used to access web applications, but not to customize the accessed web applications. Starting in 1996, Netscape introduced the *Netscape Plugin Application Programming Interface (NPAPI)* [NPAPI, 2012 and Oliphant, 1996] in the Netscape web browser. Since 2004 all renowned web browsers use this API to integrate plugins for activate content, e.g., to view PDF documents, or to play music, video, and flash content.

Academic Component Technologies

In 1998, the Department of Distributed and Dependable Systems of the Charles University in Prague published the component system *SOFA* [Hnetynka and Plasil, 2006; Bures et al., 2006; Bures et al., 2007]. SOFA is a system for building distributed component-based applications. The component model is hierarchical, i.e., a component can consist of a set of other components, either primitive or composite. Primitive components are programmed, whereas composite components are declaratively composed from other primitive or composite components.

Since 2006, the Christian Doppler Laboratory for Automated Software Engineering of the Johannes Kepler University in Linz developed the *Plux* component system [Wolfinger, 2010]. Plux is a plugin platform for extensible and customizable desktop applications. It supports dynamic composition and thus enables developers to build applications where users can load and integrate just those components they need for their current work. Users can also reconfigure an application on-the-fly by swapping sets of components while the application is running. From 2006 to now, Plux was enhanced with various improvements on the component model, as well as a solution for testing and debugging dynamically composed Plux applications. For detailed information about the evolution of Plux see the project history in Section 1.4.

2.1.2 Distribution Technologies

Berkeley Sockets

In 1970 *Berkeley Sockets* [Tanenbaum and Van Steen, 2007 pages 141-142] enabled developers to send data from one computer to another through a standardized interface. The socket interface abstracts from the underlying operating system and is a communication endpoint to which an application can write data that is sent over a network to a remote socket, where an application can read the transmitted data. Today, sockets are still base communication technology in all web systems.

Remote Procedure Calls (RPC)

In 1976, the first idea of *Remote Procedure Calls (RPC)* was published in [White, 1976]. In 1984, Andrew Birrell and Bruce Nelson published further work on RPC [Birrell and Nelson, 1984] and implemented the first version of RPC. RPC allows developers to call

procedures on remote computers instead of using explicit message exchange for remote interaction. The caller of a procedure is blocked until the receiver of the call has finished executing the procedure and the result is returned to the caller. The message passing, which is used behind the scene, is hidden from developers. In 1995, the Network Working Group released a proposed standard for the RPC Protocol Specification Version 2 [Srinivasan, 1995], which was updated by the draft standard for RPC Protocol Version 2 [Thurlow, 2009] in 2009. The idea of RPC was extended by other distribution technologies such as remoting.

Java Remote Method Invocation (RMI) / .NET Remoting

In 1997, Sun Microsystems released the *Java Development Kit (JDK) Version 1.1* [Oracle, 2013a] including the *Remote Method Invocation (RMI)* [Oracle, 2010] technology. Java RMI provides support for distributed objects and is the object-oriented successor of RPC. It allows calling methods of distributed objects with Java objects as arguments and as return values. In 2000, Sun Microsystems released the *Java 2 Platfrom Standard Edition (J2SE) 1.3* with an adapted RMI implementation that is compatible to the CORBA [OMG, 2012] standard. Since 2004, *J2SE 5.0* facilitates the implementation of distributed applications by automatic stub generation for distributed objects. Today, Java RMI is used for building distributed Java applications as well as for building communication infrastructure of higher-level component technologies such as OSGi Remote Services [OSGi Alliance, 2012b] or SOFA [Hnetynka and Plasil, 2006; Bures et al., 2006; Bures et al., 2007].

In 2002, Microsoft released the *.NET Framework 1.0* [Microsoft, 2012g] with support for *.NET Remoting* [Microsoft, 2011a]. Similar to Java RMI, .NET Remoting provides an infrastructure for distributed method calls between objects. In contrast to Java RMI, .NET Remoting not only supports the transfer of serialized object copies or proxy interfaces that forward calls to the original object on the remote computer, but .NET Remoting also supports reference identity and data synchronization for special objects that inherit the *MarshallByRef* base class. MarshallByRef objects do not only provide the interface on the remote computer, but also their data fields. Modifications on the fields of such an object get synchronized with its counterpart on the remote computer.

Web Services

In 1998, the *World Wide Web Consortium (W3C)* introduced *Web Services* [Booth et al., 2004; Haas and Brown, 2004]. Web services are a platform-independent API for components, which can be used to build distributed component-based applications. Web services communicate using XML-based protocols, e.g., *SOAP* [Gudgin et al., 2007] or *XML-RPC* [Winer, 1999]. The interface of a web service is described using the *Web Service Description Language (WSDL)* [Christensen et al., 2001]. Web services that are documented with WSDL can be registered in a public repository and discovered using the *Universal Description, Discovery, and Integration (UDDI)* [Clement et al., 2004] service.

In 2004, the *WS-interoperability Organization (WS-I)* [OASIS, 2013], which is an industry consortium, created common guidelines for interoperability of web services. WS-I published various bundled web service specifications, which are called *WS-I Profiles*. The *WS-I Basic Profile* [Chumbley et al., 2010] comprises conditions for web service implementations to be WS-I conformant, message specifications, specifications for service description, publication and discovery, as well as security specifications. Besides the WS-I Basic Profile, the consortium published a number of further WS-I profiles such as for example the *WS-I Attachments Profile*, the *WS-I Basic Security Profile*, or the *WS-I Reliable Secure Profile*.

Windows Communication Foundation (WCF)

In 2006, Microsoft released the *.NET Framework 3.0* including the *Windows Communication Foundation (WCF)* [Microsoft, 2012i]. WCF provides a unified programming model for Microsoft's distribution technologies, targets interoperability across platforms, and is used for building service oriented applications. Clients can use different transport protocols for consuming a service. WCF supports *SOAP* [Gudgin et al., 2007] over *HTTP* [Fielding et al., 1999], but also other messages over *TCP* [Postel, 1981], *Named Pipes* [Microsoft, 2013a], or *Microsoft Message Queues* [Microsoft, 2012f], which can be encoded as text or as compact binary format. Furthermore, developers can use their own transport and encoding format, if required. WCF supports reliable messaging, message queues, durable messages, and transactions. In 2006, WCF was enhanced with support for the *JavaScript Object Notation (JSON)* [Crockford, 2006] format to enable *AJAX* [Garrett, 2010] web pages for consuming WCF services. In 2010, the release of the .NET Framework 4.0 added support for *Really Simple Syndication* (former *Rich Site Summary) (RSS)* [Winer, 2003] and for *Representational State Transfer (REST)* [Fielding, 2000] to WCF.

2.1.3 Web Technologies

The World Wide Web was invented by Tim Bernes-Lee in 1990. He developed the *HTTP* protocol [Fielding et al., 1999], universal resource identifiers, as well as *HTML* [Berjon et al., 2013]. Furthermore, he developed a web server and web browser. The following sections present the history of technologies required for the Web.

Common Gateway Interface

In 1993, a team at the National Center for Supercomputing Applications (NCSA) specified the *Common Gateway Interface (CGI)* [Robinson and Coar, 2004]; a standard by which a web server can execute a program using input data that is sent by the web browser. The result of the program is passed back to the web server and sent as response to the web browser. Today, many large high-performance web sites still use web systems based on the idea of CGI.

Server Side Includes

Between 1993 and 1995 (the exact date and author could not be established) Server Side Includes (SSI) were invented. SSI is a standard that provides support to embed dynamic content into HTML documents. It replaces placeholders in HTML with values retrieved from environment variables. Placeholders can also be filled from SSI commands that provide content, e.g., from external documents or programs. Today, SSI is outdated and replaced by other technologies.

PHP: Hypertext Preprocessor

In 1995, Rasmus Lerdorf presented the *PHP: Hypertext Preprocessor (PHP)* [The PHP Group, 2012]. PHP is an open source server-side scripting language designed for building dynamic web applications. Instead of starting external processes like in CGI, PHP embeds scripts into HTML documents and interprets them on the web server. In contrast to SSI that can only embed values from environment variables or simple functions, PHP scripts benefit from a feature-rich programming language and a comprehensive web support library.

Java Web Technologies

In 1995, Sun Microsystems introduced *Java Servlets* [Mordani, 2009], a server-side Java implementation with web capabilities such as parameter support and sessions. Servlets are Java classes, which are instantiated in a servlet container on the web server in order to handle client requests. The response contains dynamic content which is produced by Java code. Thus Servlets are the Java pendant to CGI scripts. In 1995, Sun also introduced *Java Applets* [Oracle, 2013e], which are Java applications that are deployed on a web server, retrieved via a web browser, and executed in a web page on the client. Java applets are executed in a sandbox and thus have no access to local resources, such as the file system, if the permission is not granted by the user. In 1999, *JavaServer Pages (JSP)* [Delisle et al., 2006] was released as a further improvement for dynamically generated web pages. JSP allows embedding Java code and JSP commands into HTML documents. A JSP compiler translates such HTML documents into a Servlet that contains Java code. The same year, Sun Microsystems published the *Java Enterprise Edition (J2EE)* [DeMichiel and Shannon, 2013]. J2EE includes several technologies that are relevant for web application development, such as components for server applications called *Enterprise JavaBeans (EJB)* [Vatkina, 2013], *Java Database Connectivity (JDBC)* [Anderson, 2006], *Remote Method Invocation (RMI)* [Oracle, 2010], and support for web services [Booth et al., 2004; Haas and Brown, 2004]. In 2004, *JavaServer Faces (JSF)* [Burns, 2013b] was introduced, a Java-based web application framework for web user interfaces. With JSF user interfaces are programmed with web controls in an object-oriented manner using the Model-View-Controller pattern. At run time, the framework renders the web controls into HTML.

Microsoft Web Technologies

In 1996, Microsoft presented a successor technology of Server Side-Includes (SSI) called *Active Server Pages (ASP)* [Microsoft, 2012j]. ASP supports dynamically generated web pages, by embedding VBScript code into HTML documents, which is interpreted by the web server. ASP provides support for retrieving parameters from the request, for sessions, and for database connectivity. In 2002, Microsoft released an extended version called *ASP.NET* [Microsoft, 2012k] as part of the .NET Framework. As ASP.NET is built on the *Common Language Runtime* [Microsoft, 2012l], developers can use any of the languages support by .NET, such as C#, Visual Basic, and C++. In contrast to classic ASP, ASP.NET is compiled instead of interpreted, it is object-oriented, supports web controls, remoting, and web services. In 2007, the Silverlight technology [Microsoft, 2011b] was released. Silverlight is a framework for building rich web applications that support interactive media. The runtime environment is installed as a browser plugin on the client side and allows executing .NET assemblies, which are downloaded from a web server on demand, when a web application is accessed. In order that access to local resources, e.g. the file system, is possible only if the user grants the permission, Silverlight assemblies are executed in a sandbox.

Portlets

In 2003, Portlets [Abdelnur and Hepper, 2003], which allow building extensible web applications, were introduced. The Portlet API is a thin layer on top of Java Servlets [Mordani, 2009] for integrating user interface components into web applications. Portlets can communicate with each other and are integrated into a web site using a portal server.

2.2 Terminology

This section covers the terminology that is used in this thesis. The terminology is organized by technologies, i.e., terminology for component systems, terminology for distributed systems, and terminology for web systems.

2.2.1 Component Terminology

Current component systems comprise similar artifacts but use different terminology. To establish a common language, the following sections explain the terminology from the literature and compare it to the terminology of current component systems.

Software Component

Heineman and Councill [2001], Weinreich and Sametinger [2001], as well as Szyperski [2002], unanimously define a software component as a software element that conforms to the standards specified by a component model with the following characteristics: it describes its functionality through clearly defined interfaces; it is loosely coupled to

other components, i.e., it identifies other components by their interfaces but not by their implementation; the implementation is hidden, i.e., it implements an abstract data structure or an abstract data type; it is designed and packaged for third-party reuse, i.e., it can be independently deployed and composed with other components without modification.

According to Weinreich and Sametinger [2001] components can be implemented in an object-oriented way, either as a single class or by multiple classes, or in a non-object-oriented (procedural) way. For object-oriented systems, they define the term *component instance* as an object that is an instantiated component.

Component Model

A component model defines a set of standards for component implementation, namely the interface, metadata, interaction, and composition standard. The interface standard declares how components need to declare their provided and required functionality. The metadata standard describes how metadata such as the name of a component can be specified and obtained. The interaction standard specifies how components can communicate with each other. The composition standard defines how components can be composed to create a larger structure and how components can be substituted. Furthermore, a component model defines how components must be packaged so that they can be deployed independently; it defines the executable software elements that are required to execute the components conforming to the model as well as the coding conventions and documentation standards.

The following sections explain the standards of a component model in more detail, according to Heineman and Councill [2001] and Weinreich and Sametinger [2001].

Interface Standard

The interface standard defines how a component's behavior can be described by means of interfaces, non-functional specifications, and documentation. The elements of an interface are the names of operations, their parameters with valid parameter types, as well as optionally other elements, such as preconditions, postconditions, exceptions, and attributes that specify further interoperability constraints, e.g., a thread model or remoting capabilities. Many component models use an interface definition language (IDL) to describe interfaces and their elements. Interfaces serve as a contract between components, i.e., components use interfaces to specify the required and provided services. Components may specify required interfaces as mandatory, i.e., if this requirement is unsatisfied, the component itself may not be able to provide certain services. A component model can require that every component must provide a set of specific interfaces in order to conform to the component model.

Metadata Standard

The metadata standard defines how information about interfaces, components, and their relationships is specified and how it can be obtained. The metadata of

components is used for composition, scripting and reflection. Typically a metadata standard includes a naming standard that defines a schema for globally unique names for interfaces and components. Many component systems use, for example, globally unique identifiers (GUID) or hierarchical namespaces to avoid name clashes.

Interaction Standard

The interaction standard (also referred to as interoperability standard) specifies how two or more components can communicate and exchange data, e.g., it specifies the calling conventions, or how control is shared in communication channels, either within a single process, between processes on a single computer, or across multiple computers on a network. For distributed communication, the interaction standard defines common data representation and invocation semantics. It may standardize network protocols used for communication among components, e.g., SOAP, Remote Method Invocation (RMI), or Internet Inter-Orb Protocol (IIOP). Furthermore, the interaction standard covers a component's context dependencies, e.g., dependencies on other components, the operating system, or hardware resources such as a network connection.

Composition Standard

The composition standard defines how components can be composed and how already composed components can be replaced by other components. Composition is the process of combining two or more components so that components can interact with each other. Component reconfiguration means to recompose a program by adding, deleting, or replacing a component. Both, composition and reconfiguration can be done using general-purpose programming languages, scripting or glue languages, visual programming, composition tools, or component infrastructures.

Packaging and Deployment Standard

The packaging and deployment standard defines the structure and semantics for deployment descriptions. A component is deployed when it is installed and configured in a component infrastructure. It must be packaged with all required artifacts that will not exist in the component infrastructure, i.e., the code, configuration data, other components, and additional resources. The deployment description lists the contents of the package.

Component Model Implementation

The component model implementation comprises executable software elements that are required for execution of components that conform to a component model. It is typically a thin layer on the top of an operating system and must provide dedicated services for retrieving the metadata of components and registering the interfaces of components in an interface repository. For component development, it must provide tools for defining interfaces (e.g., an IDL compiler), for defining metadata, and for packaging components.

Customization

Customization means that a consumer adapts a component prior to its installation or use. Components can be customized, for example, with a deployment tool that either changes the component's metadata or with interfaces, which the component offers for this purpose. [Weinreich and Sametinger, 2001]

Extension, Host, and Contributor

Wolfinger [2010] defines an *extension* as a component that can be plugged into some other component thus extending the other component's behavior. An extension can be a *host*, a *contributor*, or both. An extension that integrates another extension (i.e., requests a service) is called a host, and the extension that is integrated (i.e., provides a service) is called a contributor.

Composition Model

Wolfinger [2010] defines the *composition model* as the subset of the component model that is responsible for discovering components, composing them, and maintaining them in the composition state. *Discovery* is the process that detects components and retrieves their metadata. *Composition* is the process that connects components, thereby matching requirements and provisions. This can be done automatically, i.e., the composition model defines how it automatically connects components based on the request specified in the metadata, or it can be done programmatically, i.e., the hosts use a composition API that is specified in the component model to find and connect their contributors manually. The *composition state* holds the instantiated components as well as their connections.

2.2.2 Distribution Terminology

The following sections define the terminology used in distributed systems. This terminology is used to describe the capabilities of existing distribution technologies in the following section and to describe Plux for Web.

Distributed System

According to Tanenbaum and Steen [2007] a *distributed system* is a collection of independent computers that appear to the user of the system as a single computer, i.e., although the computers are autonomous hardware entities, the software makes them appear as a single computer to the user.

Server / Client

In a distributed a system, computers can have different roles, i.e., a computer can either act as a server or as a client. A *server* is a computer, which provides a certain service for one or multiple clients. A *client* is a computer, which uses a service provided by a server. One computer can have both roles at a time, i.e., it can act both as a server and

as a client. The interaction between a server and a client is called request-reply interaction. [Tanenbaum and Van Steen, 2007]

Distribution Transparency

Even though a distributed system is executed on multiple computers, it appears to users as if it would be executed on a single computer. Distribution transparency means that the distribution is hidden from the developer, too. It can be achieved on different levels: if a distributed system is location transparent, developers need not care about where the remote computer is actually located; if it is access transparent, developers can access services in a uniform way, regardless of whether the service is local or remote; if it is implementation transparent, local and remote services can be implemented in the same way, i.e., no further programming effort is necessary to make a service remotely accessible.

Distributed Object, Remote Object, Serialized Object

In a program that is implemented in an object-oriented manner data and operations are encapsulated in objects, and the operations are made accessible via interfaces. In a distributed system, some objects are distributed, i.e., they do not reside on a single computer, but are transmitted from one computer to another. Distributed objects can either distribute only their interface or both their state and their interface. An object which only distributes its interfaces is called a *remote object*. With a remote object the distribution aspects are hidden behind the interface, i.e., its implementation resides on the computer where the remote object lives, while other computers make calls through the distributed interface and calls are forwarded to the original object. An object that distributes its state as well as its interfaces is called a serialized object. With a serialized object the state is copied from the original computer to the remote computer. Method calls on the remote computer are executed locally on the copied object. As changes of the object's state do not affect the original object, the developer must synchronize the changes back to the original object. [Tanenbaum and Van Steen, 2007]

Object Data Synchronization

The state of serialized objects is replicated on multiple computers. Object data synchronization is the mechanism that keeps the state of a serialized object consistent across multiple computers. On each modification of an object's state, object data synchronization transmits the changes to all other computers and updates the object states there.

Object Reference

An object reference identifies a distributed object. It typically includes the network address of the computer where the object resides and an object identifier. Object references allow clients to bind to distributed objects. [Tanenbaum and Van Steen, 2007]

Reference Identity

On remote method calls, reference identity ensures that object references which are identical in the original environment of the object, are also identical in the remote environment, i.e., if an object is transferred to a remote environment multiple times, the remote environment always gets a reference to the same object. Vice-versa, if the object is transferred back from a remote environment to the original environment, the reference to the original object is used in the original environment. Depending on the system, this can be ensured for remote objects, for serialized objects, or for both.

Lifetime Management

Lifetime management constitutes policies for creation and destruction of distributed objects. Simple approaches are, for example, to create an instance of a distributed object for each call and destroy it after the result is passed back to the caller, or to apply the singleton pattern to a service and create a single shared object at startup time and reuse the same shared object for each call until the end of the program. If the distributed system supports sessions, shared objects can be used within a session to provide a separate object per user session.

If the lifetime management pursues the same lifetime policies for distributed objects as for local objects, implementation transparency can be achieved. For example, if reference counting is used for local objects, distributed reference counting must be used for distributed objects; if local objects are garbage-collected, distributed garbage collection must collect distributed objects. To connect reference counting or garbage collection across multiple computers requires coordination, e.g., by periodically pinging other computers whether a remote object is still in use.

Thread Management

Thread management constitutes policies with respect to which threads are used for executing the methods of distributed objects. Simple approaches are, for example, using a single thread to execute all method calls; assigning a dedicated thread for every distributed object; or acquiring a new thread from a thread pool on every method call. For implementation transparency, the thread management must ensure the same threading policy for local method calls as for remote method calls. [Tanenbaum and Van Steen, 2007]

Object Server

An object server maintains object references and the corresponding instances of distributed objects. It manages the life-cycle of distributed objects and the threading of method calls to them. Service providers register their distributed objects in the object server, clients retrieve them. [Tanenbaum and Van Steen, 2007]

Fault Tolerance

As network communication is unreliable, remote method calls are more prone to failures than local method calls. Fault tolerance is the characteristic by which a system can mask the occurrence and recovery from failures. A system is fault tolerant if it can continue to operate in the presence of failures. For example, a fault tolerant system can reconnect to a remote computer after a previous failure and complete a transaction. [Tanenbaum and Van Steen, 2007]

Security

Security in distributed systems concerns authorization as well as secure communication between remote computers. The communication is secure if the sender and the receiver communicate through a secure channel so that only they can access the transferred messages. Authorization ensures that a remote computer gets only those access rights to which it is entitled. [Tanenbaum and Van Steen, 2007]

2.2.3 Web Terminology

The following sections define the terminology used in web applications. This terminology is used to describe the capabilities of existing web technologies in the following section and to describe Plux for Web.

Web application

A web application is a program that is executed on a web server and accessed by a web browser on a user's computer. The user opens a web application by entering the web address of the application in the web browser. The web browser generates a request (see below) and sends it to the web server. The web server processes the request and generates a response (see below), which contains HTML (and possibly JavasScript) that is sent back to the web browser to be rendered there. Web applications can be accessed by multiple users from different computers at the same time.

Request

A request is a message sent from a web browser to a web server containing a command with parameters to be executed by the web application as well as additional information, such as user identity or browser information. A request is the input that is processed by a web application.

Response

A response is a message that is produced by a web server as an answer for a request. The response contains the result that was generated from the web application and contains HTML (and possibly JavaScript), which is sent back to the client-side web browser to be rendered there.

Round Trip

A round trip is the sequence of sending a request to the server, processing it by the server-side web application, and replying the response back to the web browser. Round trips are triggered by users of a web application either when they enter a web address in the web browser or when they click on a button or on a link. Round trips can also be triggered by client-side script code that is executed by the web browser.

Session

A session is an interaction sequence between a web browser and a web application with related round trips. In principle, every round trip is independent from other round trips. However, web servers can assign consecutive round trips to sessions, in order to allow web applications to maintain individual application state per user.

2.3 Evaluation of Existing Technologies

Web applications that are component-based, customizable per user, and distributed across multiple computers, require the following capabilities: componentization, customization, distribution, multi-user, and web support. Existing systems typically cover only a subset of these capabilities. Component systems, for example, cover componentization and customization. However, they do not offer multi-user support. Web systems, on the other hand, cover multi-user support and web support, but do not offer customization. Section 2.3.1 describes the relevant capabilities in detail, Section 2.3.2 evaluates which capabilities are supported by existing technologies, and Section 2.3.3 addresses their remaining deficiencies for building component-based distributed multi-user web applications.

2.3.1 Relevant Capabilities

This section dissects the relevant capabilities into individual features, and describes why those capabilities are useful for building user-customizable web applications that can be distributed across multiple computers.

Customization Capabilities

Customization means to adapt features of an application to meet specific requirements of individual users. Depending on the capabilities of a system, different stakeholders can customize an application: the *developer* can always customize it, because he can modify the source code and recompile it; the *deployer* can customize it, if the system is configurable, e.g., by using a configuration file; and the *end user* can customize it, if the system provides a configuration mechanism that is suited for end users, e.g., in the user interface.

A web application follows the client-server paradigm, i.e., the application is executed on a server and is accessed through a web browser that is executed on the client. Thus

customization is feasible *on the server* or *on the client*. Depending on a system's customization capabilities, the application can be customized on the server, on the client, or on both.

A web application can be customized either by changing its *configuration* (e.g., by changing a value in a configuration file) or by modifying its *composition* (e.g., by replacing a component), if it is component-based. Typically, composition offers more flexibility than configuration, because it allows a component to be completely replaced with another one or to be extended by new components.

Depending on its architecture, a web application can be customized *in all parts* or only *in predefined parts*. With a component-based architecture, the application can be customized in all parts that are implemented as a component. In contrast, with a monolithic architecture, the developer needs to explicitly define customization points to make a certain part of the application customizable.

Customization can take place at different times during the life of a web application: it can happen at *development time*, at *deployment time*, at *startup time*, and at *run time*. The later the customization should be performed, the more flexible the mechanism must be. Of course, the developer can customize the web application by modifying the source code at development time. At deployment time, the administrator can only customize the web application, if the developer provides a mechanism for that, e.g., a setup routine with customization options. At startup time, the web application can be customized, if it retrieves configuration from a source that can be modified by the administrator, e.g., a configuration file. Customizing a web application at run time demands the most flexible mechanism, because the web application must react to configuration changes while it is running and must adapt itself accordingly.

Distribution Capabilities

Distribution means that a web application is executed on multiple computers, while it appears to the user as if it were executed on a single computer. Depending on the distribution capabilities of a system, the developer must consider distribution to a less or greater extent, e.g., he might have to implement remote components in a different way than local components. The more a system abstracts distribution, the more reusable the components can be, because distribution aspects do not have to be programmed, but are provided by the system.

A web application can be distributed *across multiple server* computers and possibly even *across client computers*. In the web context, we define a web client as a computer that requires only a web browser as an infrastructure. Following that definition, we distinguish between systems that can distribute the web application only across server computers (e.g., running a web server and a database server) and systems where parts of the web application can also run on the web client.

A distributed system can provide multiple layers of transparency: *location transparency*, *access transparency*, and *implementation transparency*. The more transparent a system is,

the less distribution must be considered by the developer: if a system is location transparent, the developer can ignore on which computer a component is executed; if it is access transparent, he can do local and remote calls uniformly; and if it is implementation transparent, he can implement local and remote code in the same way.

A distributed system can provide more or less support for managing the lifetime of distributed objects. Distributed objects occur either as dedicated service objects or as general objects that are passed as arguments. In general, the more capabilities a system provides, the less a developer must do. We distinguish the following levels of lifetime management: with *manual lifetime management*, the developer himself must implement a mechanism for creating, retaining, and releasing objects; with *semi-automatic lifetime management*, the system provides support for such a mechanism (e.g., reference counting), which the developer can use; with *automatic lifetime management*, the system has such a mechanism (e.g., distributed garbage collection) built-in and the developer does not need to care.

The fact that a distributed system is executed on multiple computers, but should appear as if it were executed on a single computer, requires further support, if objects are passed across computer boundaries. An object that is transmitted from one computer to another exists twice. However, it should appear as the same object on both computers. One transparency aspect is reference identity, i.e., if the same object is transmitted twice, it must have the same reference on the target computer both times. We distinguish systems by their support for *reference identity of dedicated service components*, *remote objects*, and *serialized objects*. A further transparency aspect is *data synchronization* of transmitted objects. Without synchronization, a change to a transmitted object on the target computer, does not affect the original object on the source computer. Thus changes need to be transmitted back to the source computer manually. Systems can support synchronization in two ways: either a proxy is transmitted as a remote object that forwards calls immediately to the original object on the source computer and thus changes affect the original object at once; or the object is serialized to the target computer and changes are automatically synchronized back to the original object by the system.

As distributed systems are executed on multiple computers, already a single-threaded application is executed by multiple threads, namely by a separate thread on each computer. However, for implementation transparency, it should appear single-threaded and the *thread management* must ensure that only one computer is active at a time, i.e., that only one thread is executed at a time, and that each thread makes synchronous calls. If repeated calls are made from the same thread on the source computer, they must all execute on the same thread of the target computer, e.g., to support thread-local data. Furthermore, if remote objects are passed across computer boundaries, calls coming back from the remote side must be dispatched in the caller's original thread. For multi-threaded applications, the thread management must provide the same support as for single-threaded applications, but for each thread individually so that each thread on a computer has a counterpart on the other computers.

In contrast to applications that are installed on a single computer, for distributed systems a decision has to be made as on which computer the components should be installed. Depending on the system, this decision can be made either by stakeholders, i.e., *administrators* or *end users*, or by the *infrastructure* of a distributed system. Administrators can decide whether they want to deploy a component to one of the server computers or to one or several client computers. End users can decide whether to install a user-specific component just on their own client computer (if they need it just locally) or to install it on the server (if they also need to use it from other client computers). For components, which are installed on the server, but should be executed on the client computer (e.g., rich user interface components), the infrastructure is responsible for downloading them to clients on demand.

The support for transparency in a distributed system determines how freely parts can be distributed across computers: with full transparency, *any customizable part* can be installed on any computer; with limited transparency, *defined parts* that belong to the same subsystem must be installed on the same computer.

In distributed systems with support for *interoperability*, components can be implemented in different programming languages (e.g., C++, Java, or C#) and still be integrated into a seamless web application.

Multi-user Capabilities

A multi-user web application is a program that is executed on one or multiple computers and is shared by multiple concurrent users, where each user has its own data that is separated from the data of other users. Furthermore, a multi-user web application can appear or behave differently for each user.

Data, appearance, and behavior can be influenced by *user-specific state*, *user-specific configuration*, or *user-specific composition*. The user-specific state contains the user's data (e.g., the shopping cart in a web shop). It can change the appearance (e.g., highlighting items with long shipping time), and it can change the behavior (e.g., paging depending on the number of items in the shopping cart). User-specific configuration can change the appearance (e.g., display more or less detailed item descriptions) or change the behavior (e.g., 1-click-checkout vs. checkout with explicit confirmation). User-specific composition can change the appearance structurally (e.g., grid control instead of a list control) or change the behavior (e.g., modified or additional business logic).

Customizations in a multi-user web application can affect either *individual users* or *groups of users*. A user can belong to multiple groups and user groups can be organized hierarchically. With *hierarchical user groups*, a group can inherit the customizations from one or multiple parent groups, i.e., the groups are organized as a directed acyclic graph, where each group inherits the customizations of its parent groups.

A multi-user web application can be customized (i.e., its appearance or behavior can be changed) by the following stakeholders: the *developer* can program a customization (e.g., to differentiate between different world regions); the *administrator* can customize

the configuration or deploy different components for users or user groups (e.g., to differentiate between managers and workers); and the *end user* can customize his configuration or deploy private components to his composition (e.g., to include his personal address book).

Web Capabilities

A web application is characterized by the facts that it is executed on a web server, that clients access it with a web browser, and that is supports multiple concurrent clients. Web applications are based on the request-response pattern, i.e., the web browser sends a request message to the web server, the web server forwards the request to the web application, the web application processes the request and generates a response message, and finally the web server sends the response back to web browser. Web support means to provide the capabilities that are required to implement a web application.

The client-side web browser must communicate with the web application. Without *communication support*, a developer must encode messages before they are sent, and decode them when they are received. With communication support, the web system provides abstract means for encoding and decoding, i.e., the developer can handle commands to the web application in a similar way as method calls.

When the web application processes a request, the developer needs to retrieve the commands from the request. Without *parameter support* the developer needs to analyze the request message manually. With parameter support, the web system provides the command in a structured manner.

The response of a web application can contain static or dynamically generated content. Without support for dynamic content, the developer must insert the dynamic elements into the HTML programmatically. If the web system supports templates, the developer can create HTML *templates* with *placeholders*, which are replaced by the dynamic content. If the web system supports *embedded code*, the developer can embed method calls into the HTML, which insert the dynamic content when they are evaluated. If the web system supports *web controls*, the HTML programming logic is encapsulated in objects; the developer can use an object-oriented programming style, i.e., set property values, call methods, and listen to events; and the web system automatically renders the HTML and generates JavaScript from the web controls.

As web applications support multiple users concurrently, they must handle multiple requests at the same time. In a multi-threaded web system, each request is executed in a separate thread. Without *threading support*, the developer must manage threads manually, i.e., start and stop the threads, and manage them efficiently, e.g., by using a thread pool. If threading support is available, the web system handles the thread management.

In principle, web applications are stateless, i.e., they process every request independently. However, typical web applications maintain a state for each user, i.e.,

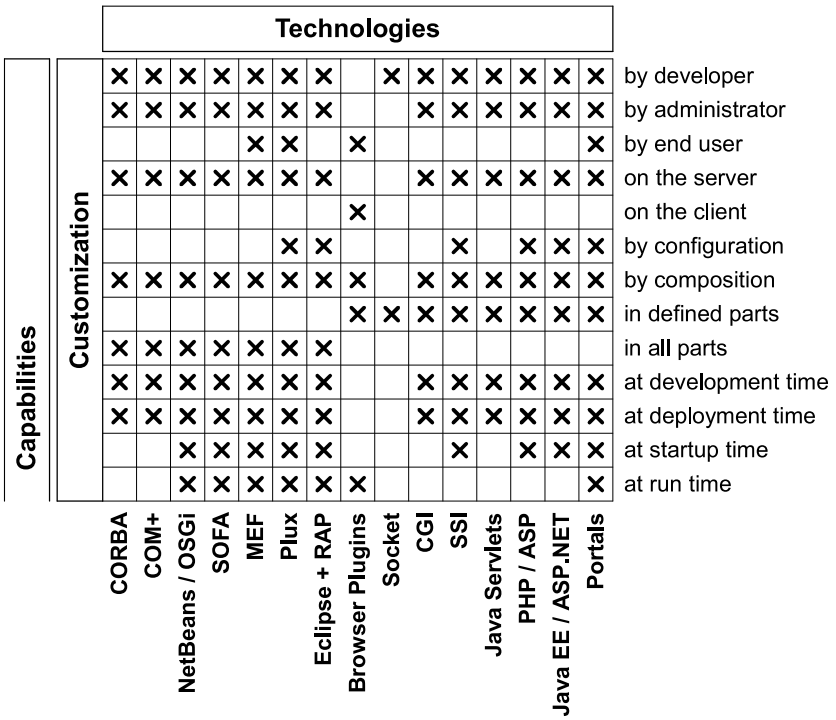
they keep user data between consecutive requests, e.g., when the user collects multiple items in a shopping cart. Web systems maintain user states in *sessions*. A session keeps track of a user's activity and is maintained as long as the user interacts with the web application or exceeds a specified timeout. A web system can store session data *in main memory*, persist it *to non-volatile storage* (e.g., to a disk or to a database), or even *move it to other computers* within a server farm, to balance the load.

2.3.2 Capabilities of Existing Technologies

This section analyzes current technologies that are relevant for building component-based distributed multi-user web applications. We analyze and compare the technologies with regard to the required capabilities described in the previous section. Figure 2.1 on page 29 and 30 shows the capabilities provided by existing technologies, while the following subsections discuss them in detail.

Common Object Request Broker (CORBA)

CORBA [OMG, 2012] is a language-independent component model that isolates service providers from service requestors by encapsulating interfaces. Interfaces of components (object implementations) are specified in the CORBA Interface Description Language (IDL). The contributors (providers) are stored in a repository and the hosts (requestors/clients) retrieve their contributors using an object request broker (ORB). An application can be customized by modifying the composition in all parts where CORBA components are used, at development time as well as at deployment time, i.e., the developer can change the source code and the administrator can configure the



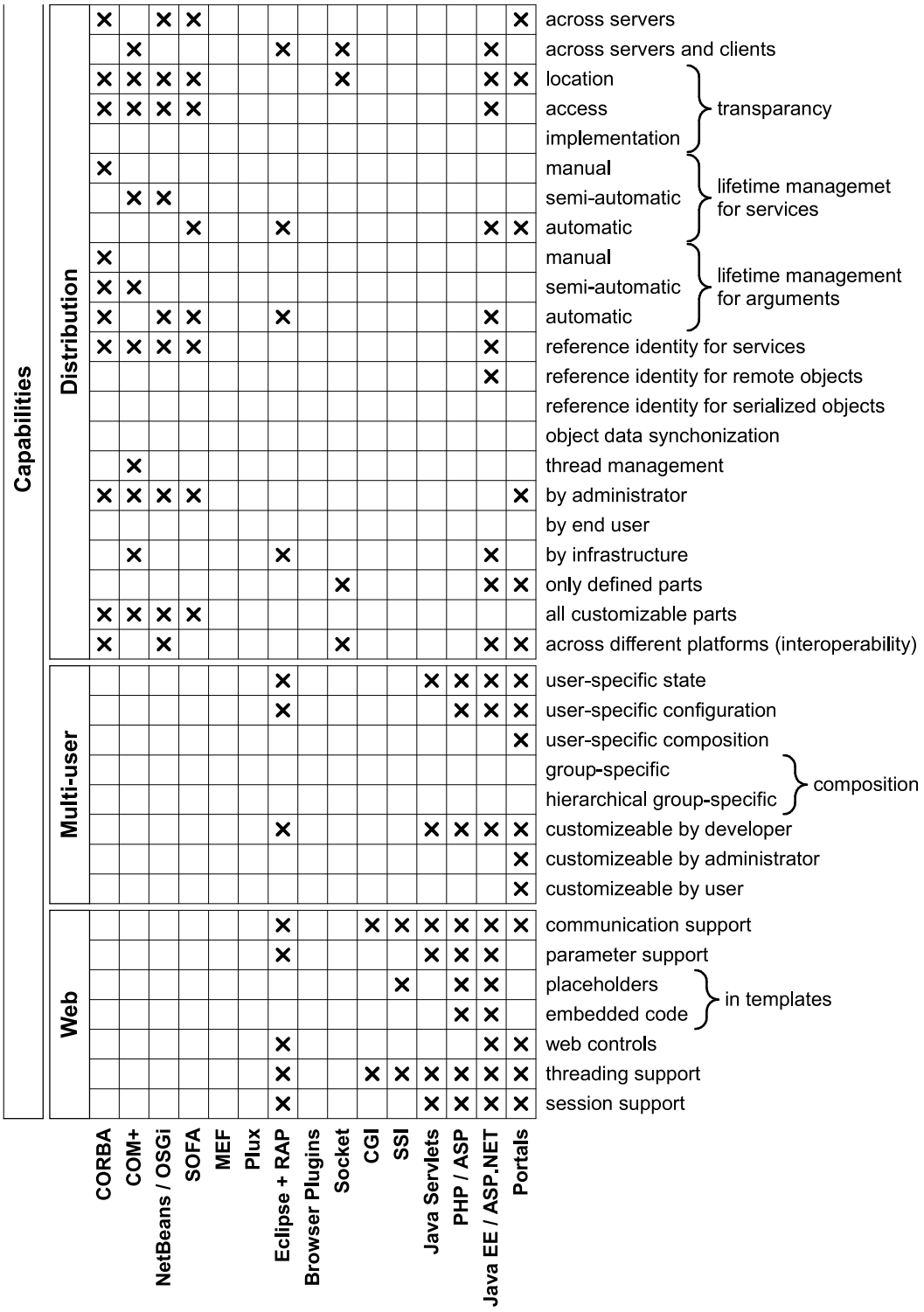


Figure 2.1: Capabilities of existing technologies

repository. End user customization capabilities are not provided. All components of a web application built with CORBA are executed on the server exclusively, therefore client-side customization cannot be provided.

The ORB core generates skeletons and stubs for components from the IDL descriptions and handles communication between components that can be distributed across multiple computers. The hosts use Object Adapters to access services from the ORB and get support for location transparency and access transparency. The life cycle of services is managed manually by the hosts. The lifetime management for objects that are transferred as arguments is depending on the technology of the implementation of CORBA, e.g., Java supports automatic lifetime management, whereas C++ supports semi-automatic lifetime management with reference counting.

CORBA supports reference identity for services, distinguishing between CORBA objects and instances of value types. A CORBA object implements an IDL interface and is registered with the ORB. Such objects can be passed as reference parameters. How other objects (value type objects) are passed, depends on the receiver. If the receiver is a CORBA object, services are passed by reference, other objects by value. If the receiver is a value type object, the semantics of parameter passing depends to the programming language in which the ORB is implemented. A CORBA object can be assigned to a computer by registering its Object Reference on the computer's ORB. Thus the distribution of CORBA objects can be controlled by administrators, but not by end users. CORBA objects stay on the computer where they are registered; the infrastructure does not redistribute objects to other computers automatically.

CORBA does not target web applications and provides no multi-user or web support.

Component Object Model (COM)

COM [Microsoft, 2012c] is a binary component standard that is available on Windows operating systems. COM components can be implemented in multiple languages, e.g., in C/C++, Visual Basic, Delphi, and C#. Interfaces of components are specified in the Microsoft Interface Description Language (MIDL). The contributors (servers) are stored in the Windows Registry and the hosts (clients) retrieve them using the COM API. As COM and CORBA are conceptually similar, their customization options are the same.

The Distributed Component Object Model (DCOM) extends COM to support communication across computer boundaries. The distribution support of DCOM goes beyond CORBA with the following capabilities:

Components cannot only interact with server-side components, but also with components that are executed on the client. Such client-side components are implemented with the *ActiveX* technology [Microsoft, 1996]. ActiveX components are stored on the server and downloaded to the client on demand. Typical ActiveX components are used to embed rich content into web applications, e.g., video, audio, or PDF documents.

COM supports life-cycle management with reference counting, provided through the *AddRef* and *Release* methods in the *IUnkown* interface, which must be provided by all components. Reference counting is used for contributors (servers) as well as for parameter values.

COM objects are organized in apartments that provide thread management. An apartment controls how many threads are allowed to enter concurrently. In a single-threaded apartment (STA) components are executed by a single dedicated thread; in a multi-threaded apartment (MTA) a set of dedicated threads execute components, thus the objects must do their own synchronization; in a thread-neutral apartment (NTA) arbitrary threads can enter and execute. A process can contain STA, MTA, and NTA objects, which can interact with each other.

COM does not provide any multi-user or web support.

Open Services Gateway initative (OSGi)

The Open Service Gateway initiative (OSGi) [OSGi Alliance, 2012a] is a Java-based component standard. Interfaces of components are specified in Java and components are called *bundles*. Contributors register their services in the *Service Registry* where hosts can retrieve them. OSGi provides the same customization options as CORBA and COM, but adds additional support for customization at startup time and at run time. Services can be added and removed in the service registry while an application is running; upon changes the service registry notifies hosts with events.

Similar to the ORB in CORBA, the OSGi *Remote Services* standard [OSGi Alliance, 2012b] allows building a distributed service registry. In order to make a service available for remote clients, a *distribution provider* creates an *endpoint* from the interface of the registered service on the contributor side. The endpoint handles the remote communication, e.g., as a web service or with Java RMI. To import the service on the host side, a distribution provider creates a proxy and registers it in the service registry of the host. Thus a host can retrieve remote services from the service registry in the same way as local services, i.e., remote services are location transparent as well as access transparent.

Similar to COM, OSGi uses semi-automatic life-cycle management for services (with the *getService* and *ungetService* APIs), but unlike COM, OSGi uses garbage collection of Java and thus the life-cycle management for arguments is automatic.

The distribution capabilities (thread management, reference semantics, and synchronization) are similar to CORBA. However, as a distribution provider for an OSGi service can create an endpoint that supports web services for communication, the service is interoperable with other platforms that are not Java-based.

OSGi does not provide multi-user or web support.

SOFA Component Model

SOFA [Hnetynka and Plasil, 2006; Bures et al., 2006; Bures et al., 2007] is a component model developed at the Charles University in Prague. The component model is hierarchical as it distinguishes between primitive and composite components. Primitive components are programmed, whereas composite components are declaratively composed from other primitive or composite components.

Components are specified using the Meta-Object Facility (MOF) standard [OMG, 2006] and communicate with each other using connectors. Connectors can use different communication technologies such as procedure calls, messaging, streaming, or communication via shared memory.

The SOFA runtime environment (SOFAnode) consists of a repository, which stores the meta-data of components and the components' implementations, and it contains a number of component containers, which provide the functionality for executing components. Component containers are called deployment docks and can be distributed across multiple computers. Thus SOFA supports component distribution. Furthermore, SOFA supports dynamic reconfiguration by adding, removing, or replacing components at run time. Even though the SOFA runtime is implemented in Java, its concepts are language-independent and can be implemented in other programming languages too.

Sofa does not provide multi-user or web support.

Managed Extensibility Framework (MEF)

MEF [Microsoft, 2010] is a component model based on .NET. Interfaces are specified in a .NET language, such as C# or Visual Basic.NET. Hosts and contributors (called *parts*) declare their provided interfaces (*exports*) and requested interfaces (*imports*) using metadata.

The components' metadata are retrieved by a discoverer (*catalog*) and stored in a registry (*container*). The MEF composition engine matches provisions and requests automatically to assemble an application. MEF's customization options are similar to those of OSGi, however, with MEF the end user can also add custom components, e.g., by copying component files to a folder in the file system. Such components are discovered by the catalog and composed by the composition engine.

MEF targets desktop applications on a single computer, e.g., Microsoft Visual Studio, and thus neither provides distribution support, multi-user support, or web support.

Plux

The following description of Plux covers the state of Plux before this thesis [Wolfinger, 2010]. As Plux initially targeted only desktop applications, it did not provide distribution, multi-user, or web support. Therefore, this description focuses on the customization capabilities of Plux only.

Plux is a component model for dynamic plug-and-play composition. The composition is done by the infrastructure and not by the components. It is plug-and-play because it happens automatically. Components (*extensions*) declare their provisions (*plugs*) and requests (*slots*) using metadata. Plux retrieves these metadata, matches provisions and requests, and connects contributors to matching hosts. The composition is dynamic because components can be added and removed at run time, i.e., Plux automatically recomposes the application without restarting it. In contrast to any other component

platform, Plux keeps track of which hosts use which contributors, i.e., it maintains a *composition state* that holds all extension instances and their connections. Hosts can retrieve their contributors from the composition state and can modify the composition using the Plux composition API. The customization capabilities of Plux are identical to those of MEF, but as Plux components contain configuration metadata in addition to composition metadata, Plux applications can also be customized by configuration.

Eclipse with Remote Application Platform (RAP)

Eclipse [Eclipse, 2006] is a component platform based on Java. Interfaces of components are specified in Java. Components are called *extensions* and connect to other extensions via *extension points*. Contributors register their services in the *Eclipse Registry* from where hosts retrieve them. Eclipse provides the same customization capabilities as CORBA, COM, and OSGi. In addition to that, Eclipse applications can be customized by configuration, i.e., by changing parameter values in the XML metadata of an extension.

Eclipse targets only desktop applications, but in combination with the Remote Application Platform (RAP) [RAP, 2012], developers can generate web applications from Eclipse desktop applications. RAP generates web controls with HTML and JavaScript from Eclipse SWT widgets. The web controls are executed on the client and communicate with business logic on the server. The distribution is limited as only the user interface is transferred to the client (automatically by the infrastructure). Other parts of the web application cannot be distributed.

As Eclipse with RAP is hosted in a Java EE server, the multi-user and web support is similar to that of the Java Enterprise Edition (see below). However, as in RAP user interfaces are implemented by the use of SWT controls, web page templates are not available.

Browser Plugins

A Browser Plugin [NPAPI, 2012 and Oliphant, 1996] is an extension that is installed in the web browser on the client. The plugin can modify a web application's user interface or display rich content, such as Adobe Flash animations or PDF documents. However, browser plugins only extend the functionality of a web browser, but do not extend the web application, which is accessed via the browser. Furthermore, browser plugins are leafs from the perspective of composition and cannot be extended themselves by additional plugins.

With Browser Plugins only the end user can customize the web application, and it can only be customized on the client. However, unlike in all other web technologies, the developer and administrator of a web application have no influence on which plugins are installed on the client.

As Browser Plugins are always executed in the web browser on the client, they have no distribution, multi-user, or web support.

Sockets

Sockets [Tanenbaum and Van Steen, 2007 pages 141-142] are a communication mechanism between processes on one or several computers. They are communication endpoints that communicate by sending and receiving byte streams, usually based on the TCP protocol [Postel, 1981] or the UDP protocol [Postel, 1980]. Programs must serialize and deserialize their data in order to transmit them via sockets.

Developers can customize an application in defined parts, by connecting these parts to different sockets. Sockets can be used to distribute parts of a web application across multiple servers and even to a web client using web sockets. As the usage of a socket is independent of the other socket's location, distribution is location transparent.

In socket-based web applications, only parts that actually use a socket can be distributed. Other parts that use local procedure calls (e.g., between components) cannot be distributed. Thus, with sockets only defined parts of the application can be distributed.

Sockets are a platform-independent mechanism, i.e., parts of socket-based web applications can be distributed across different platforms.

As sockets are only a means of communication, they do not provide multi-user or web support.

Common Gateway Interface (CGI)

The Common Gateway Interface [Robinson and Coar, 2004] is a standard for dynamically generated web pages. A web server with CGI starts a new process for each request, which takes the request's parameters as input and generates the response message as output. A CGI web application can be customized in defined parts through composition on the web server. The administrator can modify the mapping of URLs and CGI scripts at startup time. CGI does not provide support for distribution or multiple users. However it provides web support. Depending on a request's URL, the web server automatically calls the CGI script that is assigned in the configuration. Thus the CGI script does not have to handle network communication but can simply read the input from the standard input stream and write the output to the standard output stream.

Server Side Includes (SSI)

Server Side Includes [The Apache Software Foundation, 2013] are a standard for simple web templates. For each request SSI replaces placeholders in the web page template with values from environment variables or simple library functions, e.g., the current date and time. An SSI web application can be customized in the same way as CGI. In addition to that, the administrator can customize by configuration, i.e., by changing the web page templates at deployment time and by setting the environment variables at startup time. The web support of SSI goes beyond CGI, because of the web page templates.

Java Servlets

Java Servlets [Mordani, 2009] are a standard for dynamically generated web pages (similar to CGI) with scripts that are implemented in Java. Servlets can be customized exactly in the same way as CGI. Java Servlets do not support distribution. However they provide multi-user and web support. In contrast to CGI and SSI, which are stateless, Servlets can keep a user state between consecutive requests in a session. Furthermore, Servlets support HTTP parameters that can be retrieved uniformly, regardless of whether they are passed in the URL or in the message body. The user state of a session can be kept in main memory, on disk, or even on multiple servers to balance the load.

Server-side Scripting Languages

PHP: Hypertext Preprocessor (PHP) [The PHP Group, 2012] is a server-side scripting language for dynamic web pages that are rendered from web page templates. In PHP the web page templates can contain script code that is evaluated for each request and the results replace the placeholders in the template. The customization options of PHP are the same as in SSI. PHP does not support distribution. The multi-user and web support in PHP is similar to that of Java Servlets, plus the support for web page templates.

Active Server Pages (ASP) [Microsoft, 2012j] and Java Server Pages (JSP) [Delisle et al., 2006] are further representatives of server-side scripting languages with similar capabilities as PHP.

Java Enterprise Edition / ASP.NET

Java Enterprise Edition (Java EE) [DeMichiel and Shannon, 2013] is a standard for a web application framework. In contrast to server-side scripting languages, the business-logic is not embedded in the web page templates, but encapsulated into business-logic components (Enterprise Beans). Moreover, the user interface is encapsulated in web controls (JavaServer Faces [Burns, 2013b]). Web controls are reusable Java components that can be programmed in an object-oriented manner just like business-logic components. Web controls are embedded into web pages and are dynamically rendered into HTML and Javascript on each request.

The customization capabilities for Java EE web applications are similar to those of server-side scripting languages.

Java EE provides support for distribution of defined parts across servers (e.g., the database server can run on a different computer than the web server) or even across servers and clients (e.g., web controls on the client can connect to the server application using AJAX). As distributed web components are implemented with RMI or web services, they are location transparent and access transparent, i.e., developers do not have to care on which computer a component is located, and local components can be accessed in the same way as remote components. The lifetime management of service

objects and argument objects is done automatically in Java EE. If web services are used for distribution, Java EE is interoperable with other platforms.

The multi-user support and the web support of Java EE is the same as in Java Servlets, as Java Servlets are part of Java EE. In addition to that, Java EE supports web page templates (JavaServer Pages) and web controls (JavaServer Faces).

ASP.NET [Microsoft, 2012k] is a web application framework for the .NET platform with essentially the same capabilities as Java EE. However, the distribution support in ASP.NET (.NET Remoting) goes beyond Java EE (Java RMI) with respect to reference identity and object synchronization.On the other hand, .NET Remoting is not interoperable, i.e., it is limited to the .NET platform, and it requires that all remote objects inherit a special base class, whereas in Java RMI it is sufficient to implement an interface.

Portals

A Portal is a web-based application that is composed from multiple user interface parts which are embedded from other web sites. Portals are used to aggregate web content from different sources for personalized web sites. Portlets [Hepper, 2008] is a standardized Java technology for implementing Portals. A Portlet is a pluggable user interface component. It processes requests and generates dynamic HTML and JavaScript content. A Portlet container runs Portlets, i.e., it provides services such as lifetime management, persistence, and preferences for Portlets. The web page of a Portal is generated from all Portlets in the container.

The customization options of Portals are similar to those of Eclipse with RAP, but with two differences: Portals can only be customized in defined parts, i.e., Portlets can be added, arranged, and removed. On the other hand, a Portal can also be customized by end users, which Eclipse with RAP cannot.

Portlets can be distributed across multiple web servers. The distribution of Portlets is controlled by the administrator, as he installs the Portlet on a web server. Portlets are based on open web technologies, such as HTML and JavaScript, and therefore are interoperable across different platforms.

Portlets provide full multi-user support, i.e., user-specific state, configuration and composition (for users, groups, and hierarchical groups), as well as user-specific customizability by developers, administrators, and end users.

Portlets provide similar web support as Eclipse with RAP, but they cannot retrieve HTTP parameters from a user request, because Portlets are retrieved by the Portlet server with a separate request instead of the original user request.

2.3.3 Deficiencies of Existing Technologies

To develop a component-based distributed multi-user web application, developers need a technology that provides support for all relevant capabilities, i.e., support for

componentization, support for distribution, multi-user support, and web support. As this chapter shows, none of the existing technologies provides support for all of these capabilities. A possible solution is to choose a combination of technologies in order to cover as many of the required capabilities as possible. For example, a developer could combine the component technology OSGi, the web technology Java EE, and the web component technology Browser Plugins. The developer could build the component-based web application with OSGi and thus make it customizable by composition on the web server. Using OSGi remote services, he could distribute parts of the web application across multiple computers. Java EE would provide support for building the user interface and multi-user support with sessions. In order to integrate components from the web client, the developer could use Browser Plugins.

A developer that applies this strategy would have to learn and master three different technologies and would still face several limitations:

End-user customization is limited to browser plugins. Thus it is limited to a small subset of the web application's components. Furthermore browser plugins are limited in composition, they cannot integrate seamlessly with components on the web server, e.g., to access data components of the web application's backend. Finally, browser plugins need to be implemented in a different programming language as all other components.

The distribution of components is limited in several ways: client-side components cannot be moved from the client to the server or vice versa because they are implemented in different languages and for different component models than server-side components. Client-side components are implemented, for example, in C/C++ as Browser Helper Objects for the Internet Explorer; server-side components are implemented in Java as OSGi components. Furthermore, in order to change a local OSGi component into a remote OSGi component, the developer must create a distribution provider. If such a provider is not available for a component, it cannot be accessed remotely. Moreover, the developer must be aware whether he accesses a local or a remote component, e.g., because remote components have different reference semantics than local components and require different thread management and data synchronization.

The multi-user support is limited as only Java EE provides multi-user capabilities but OSGi does not. As the composition is done by OSGi, the composition is maintained for all users in common. Therefore each user must have the same composition and cannot integrate his own components without affecting other users. Only the web support is sufficient, as Java EE provides all the required capabilities.

Chapter 3

The Plux Component Model

This chapter presents the Plux component model. The metadata standard specifies how to declare components. Components are called extensions and use the metaphor of slots and plugs: extensions, which declare a slot, want to use other extensions; extensions, which declare a plug, provide a service to other extensions. The deployment standard specifies an exchangeable discovery mechanism, whereby extensions are self-contained, so that no separate configuration files are necessary for composition. The composition standard specifies how Plux provides full knowledge about the connections between extensions by maintaining a composition state, how Plux performs the composition automatically instead of programmatically, and how composition libraries can adapt the automatic composition process. The interaction standard specifies how extensions can communicate and exchange data in a dedicated runtime thread. The customization standard specifies how extensions can be configured with a common settings model.

The Plux component model specifies metadata, deployment, composition, interaction, and customization standards for extensible and customizable applications that are composed from plugins. Plux supports plug-and-play composition to compose programs from plugins without programming or configuration effort. Plux also uses dynamic composition, i.e., it allows reconfiguring a program by adding and removing components while the program is running. The Plux component model was originally published in the dissertation of Wolfinger [Wolfinger, 2010]. As we improved and extended the Plux component model, this chapter presents its current state.

Plux distinguishes itself from other plugin component models [Birsan, 2005], such as CORBA, COM+, Eclipse, or OSGi, by the following key characteristics: in Plux, a central composer automatically connects components and maintains their connections in a composition state; it uses events to notify components about changes in the composition state; components are discovered dynamically using an exchangeable discovery mechanism; components can be configured with dynamically discovered settings.

Plux replaces programmatic composition with automatic composition. In programmatic composition, the composition logic is implemented in the components

themselves. Components register their provided services in a global registry, while other components either query the registry to retrieve registered components, or they listen to change events sent by the registry, to integrate registered components programmatically. The drawback of this solution is that every host has to implement the retrieval of contributors from the registry itself. This results in coding overhead and code duplication. Furthermore, if each host implements this mechanism on his own way, the composition implementation might become inconsistent and not uniform. In Plux, the composition mechanism is not implemented in the components, but rather in a central composer of the Plux runtime, which performs the composition automatically. Automatic composition means that the components declare their requirements and provisions using metadata; the composer uses these metadata to match requirements and provisions and to connect matching components automatically. This minimizes coding effort and unifies the composition mechanism. During composition, Plux sends composition events to the affected components so that they can react.

At any time, Plux maintains the current composition state, i.e., it keeps track of which components are connected to which others, and also stores an arbitrary number of named labels on connections (which are called tags). As components can retrieve the composition state, they do not need to store references to the components they use.

Discovery is the process of detecting new components and extracting their metadata. Unlike in other plugin systems, the discovery mechanism is not an integral part of Plux, but is a plugin itself. This makes the mechanism replaceable. Components are configured by settings that are provided by the discovery mechanism, which allows reconfiguring components dynamically. The following subsections cover those characteristics in more detail.

3.1 Metadata Standard

Plux uses the metaphor of extensions that have slots and plugs (Figure 3.1). All of them are specified using metadata. An *extension* is a component that provides services to other extensions and uses services provided by other extensions. If an extension wants to use a service of some other extension it declares a *slot*. Such an extension is called a *host*. If an extension wants to provide its service to other extensions it declares a *plug*. Such an extension is called a *contributor*. Related extensions can be packaged as a *plugin* so that they can be deployed as a single unit.

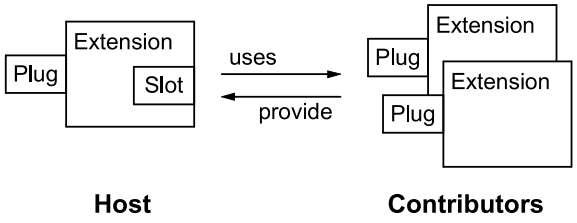


Figure 3.1: Metadata for Plux extensions with slots and plugs

Slots and plugs are identified by names. A plug matches a slot, if their names match. If so, Plux will try to connect the plug to the slot. A slot represents an interface, which has to be implemented by a matching plug (Figure 3.2). The interface is specified in a *slot definition*. A slot definition has a *unique name*, optional *parameters* whose values must be provided by the contributors and can be retrieved by the hosts, as well as optional *tags* that can be set in the composition state and can be retrieved by the hosts and the contributors. The names of slots and plugs refer to the respective *slot definition*. Multiple slot definitions can be packaged as a *contract*.

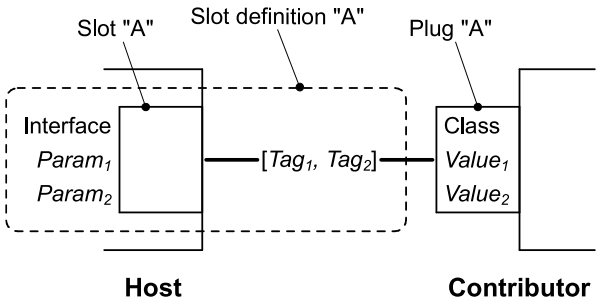


Figure 3.2: Metadata of a *slot* and a *plug* named "A" defined in a *slot definition*

The means to provide metadata is customizable in Plux. The default mechanism extracts metadata for plugins and contracts from .NET attributes in assembly files. Assembly files are DLL files that contain .NET classes, metadata, and resources. .NET attributes are pieces of information that can be attached to .NET constructs, such as classes, interfaces, methods, or fields. At run time, the attributes can be retrieved using reflection [Ecma, 2010].

Plux specifies the following custom .NET attributes (see examples from Listing 3.1 to Listing 3.3): the *SlotDefinition* attribute to declare an interface as a slot definition, the *Extension* attribute to declare a class as an extension, the *Slot* attribute to declare requirements for contributors in hosts, the *Plug* attribute to declare provisions in contributors, the *ParamDefinition* attribute to declare required parameters in slot definitions, the *Param* attribute to declare provided parameter values in contributors, and the *TagDefinition* attribute to declare optional tags in slot definitions. Plux can use arbitrary objects as parameter values, however the default metadata mechanism of .NET attributes limits parameter values to compile-time constants.

Let us look at an example. Assume that we have a workbench implemented as a host extension working with views that are implemented as contributors. The workbench displays the view's titles in its view menu and their controls within the workbench window. Figure 3.3 on the next page shows the user interface of the workbench with an email and a payroll view, as well as the corresponding extensions with plugs and slots. The *Workbench* extension is plugged into the *Application* slot of the Plux core and acts as a host for the contributor extensions *Email* and *Payroll*, which are views for the workbench that are plugged to it via a *View* slot and tagged with a *Menu* tag.

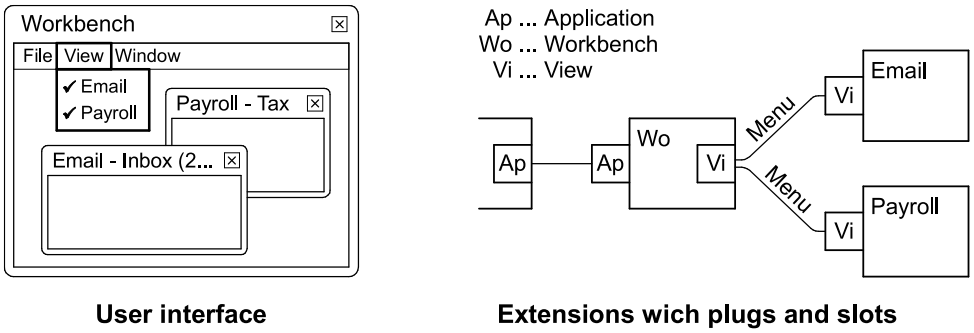


Figure 3.3: User interface and extensions of a workbench application

In order to implement this example, we need to define the *View* slot into which the views can plug, i.e., we create the interface *IView* and mark it with the *SlotDefinition* attribute (Listing 3.1). Each view contributor must provide two different titles: a static title for views that are currently not opened, and a dynamic title for opened views. The *ParamDefinition* attribute *Title* ensures that view contributors must provide a static title as a parameter value. The static title is retrieved as a parameter (and not as a part of the interface) because the workbench needs to retrieve the title without instantiating a view to show the title of available views in its view menu, e.g., "Email". The dynamic title can be retrieved with the *GetTitle* method of the interface and thus can reflect the currently displayed content in the caption of the view, e.g., "Email - Inbox (2 new messages)". To retrieve the user control of the view, the workbench calls the *GetControl* method. In order to make the menu of the workbench customizable, we use the *TagDefinition* attribute to define a tag *Menu*. Through this tag, the workbench determines whether it must show the static title of a view in the view menu: view contributors that are tagged with *Menu* are shown in the menu, others are not.

```
[SlotDefinition("View")]
[ParamDefinition("Title", typeof(String))]
[TagDefinition("Menu")]
interface IView {
  String GetTitle();
  Control GetControl();
}
```

Listing 3.1: Interface and metadata for a slot definition

Next, we implement a contributor for the *View* slot. Listing 3.2 shows a view for emails. The *Extension* attribute marks the class *EmailView* as an extension and the *Plug* attribute *View* marks it as a contributor for the *View* slot. As required by the slot definition, the class implements the interface *IView* and provides a value for the parameter *Title*.

Finally, we implement the workbench extension (Listing 3.3). To make it a host for views, we specify a *View* slot using the *Slot* attribute. As the workbench is also a contributor for the *Application* slot of the Plux core, we apply the *Plug* attribute *Application* and implement the corresponding *IApplication* interface that defines the

```
[Extension]
[Plug("View")]
[Param("Title", "Email")]
class EmailView : IView {
  String GetTitle() { /* not shown */ }
  Control GetControl() { /* not shown */ }
}
```

Listing 3.2: Implementation and metadata for a contributor extension

method *Start*. At startup, Plux creates an instance of the workbench, connects it to its core, and calls the workbench's *Start* method. The implementation of the class *Workbench* is covered in Section 3.3 Composition Standard.

```
[Extension]
[Plug("Application")]
[Slot("View")]
class Workbench : IApplication {
  void Start() { /* not shown */ }
  ...
}
```

Listing 3.3: Implementation and metadata for a host extension

To complete the example, we compile the slot definition for views (Listing 3.1) into a contract assembly *Workbench.Contract.dll*, the workbench extension (Listing 3.3) into a plugin assembly *Workbench.dll*, and the view extensions for email (Listing 3.2) and payroll (implementation not shown) into the plugin assemblies *Email.dll* and *Payroll.dll*.

3.2 Deployment Standard

Slot definitions are deployed in contracts and extensions are deployed in plugins, i.e., both are deployed in DLL assembly files. The Plux discovery mechanism detects contracts and plugins and extracts the metadata of slot definitions from a contract and the metadata of extensions from a plugin. Next, it notifies the Plux core about discovered contracts and plugins, which stores them to retrieve their metadata during the composition of the application (see Section 3.3 Composition Standard).Vice versa, the discoverer also notifies the Plux core when contracts and plugins are removed. Plux supports dynamic reconfiguration, i.e., plugins can be added and removed at run time without restarting the application.

In order to make the discovery mechanism customizable, it is implemented as an extension itself. For this, the Plux core provides a *Discovery* slot for discoverer extensions, beside the Plux core's *Application* slot for applications. The default *Discoverer* extension watches one or more directories for newly added or removed assembly files and reads the metadata from their attributes. Figure 3.4 on the next page shows an example with a *Discoverer* extension that is plugged into the *Discovery* slot of the Plux core. The *Discoverer* extension declares two slots: a *Detector* slot for

contributors, which detect newly added or removed assemblies, and an *Analyzer* slot for contributors, which extract metadata from detected assemblies. The *FilesystemDetector* contributor monitors directories in the file system and detects DLL assembly files when they are copied into a directory or when they are deleted from there. The *AssemblyAnalyzer* contributor retrieves metadata from custom attributes of detected assemblies. In addition to the *FilesystemDetector* and the *AssemblyAnalyzer*, Plux provides further detectors and analyzers, e.g., an *XmlDetector* that monitors entries in an XML file that specify the assemblies to be detected, and a *DatabaseAnalyzer* that retrieves metadata for assemblies from a database. The Discoverer extension can use multiple detectors and analyzers at the same time.

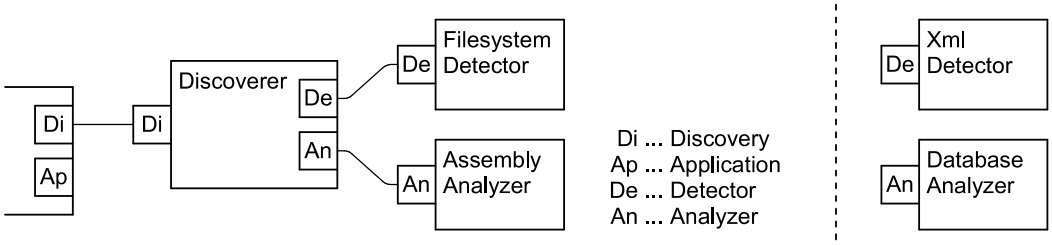


Figure 3.4: *Discoverer* extension plugged into the *Discovery* slot of the Plux core

Let us get back to the workbench example from the previous section. When we copy the contracts into a *Contracts* directory and the plugins into a *Plugins* directory (Figure 3.5 left), the *FilesystemDetector* detects the newly added files and the *AssemblyAnalyzer* extracts the metadata for the slot definitions and extensions from the assembly files (Figure 3.5 right).

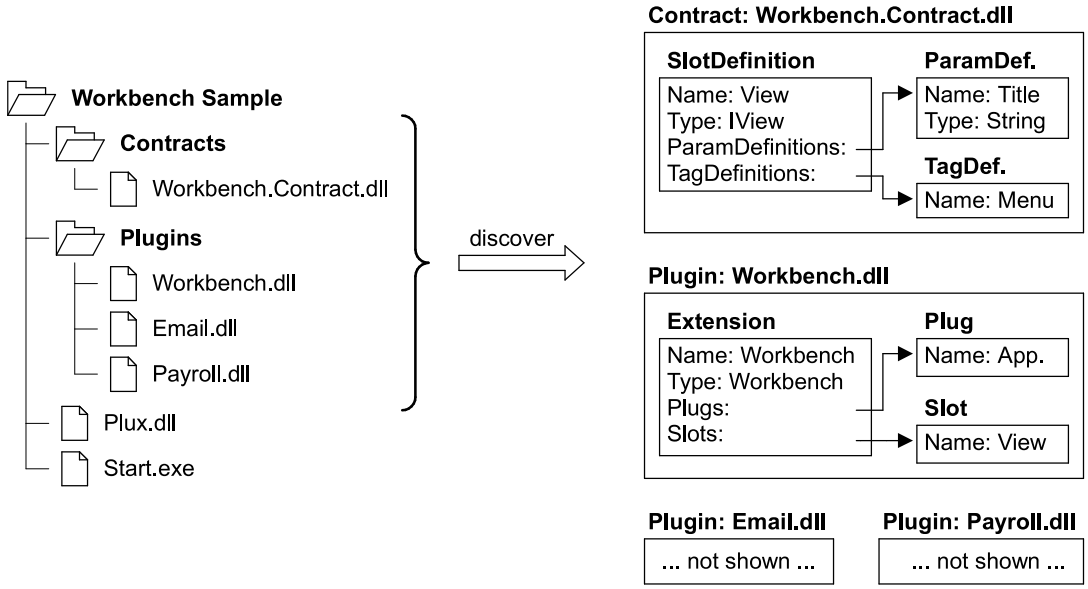


Figure 3.5: Metadata extracted by the discoverer from the contract and the plugins of the workbench application

3.3 Composition Standard

Composition is the process that matches the requirements of hosts with the provisions of contributors. In Plux, this is done by the composer, which assembles a program from extensions provided by the discoverer. When the discoverer detects a new extension, the composer integrates it into the program. Vice versa, when the discoverer detects that an extension was removed, the composer removes it from the program.

Integrating an extension means that the composer searches the composition state (Section 3.3.1) for slots that match the plugs of the new extension. If such slots are found, the composer plugs the extension, i.e., it creates an instance of the extension and connects its plugs to all matching slots in the composition state. Removing an extension means that the composer searches the composition state for slots where instances of the extension are plugged. If such slots are found, the composer unplugs the extension, i.e., it disconnects the plugs of the extension from these slots and destroys the instance (see Section 3.3.2 Composition Operations and Section 3.3.4 Automatic Composition).

3.3.1 Composition State

In Plux, all connections between components are established by the composer. Therefore the composer has full knowledge about the instantiated extensions, their slots and plugs as well as about their connections. This information is called the *composition state*. If a host wants to use its plugged contributors, it can simply retrieve them from the composition state. For every instantiated extension, the composition state holds the meta-object of the extension, the meta-objects of its slots and plugs as well as a reference to the corresponding extension object (Figure 3.6). The extension object is the extension's implementation, i.e., an instance of the class that was marked with the *Extension* attribute. For every slot, the composition state keeps track which plugs are connected to it (retrievable via the *PluggedPlugs* property); and for every plug the composition state keeps track of to which slots it is connected (retrievable via the *PluggedInSlots* property).

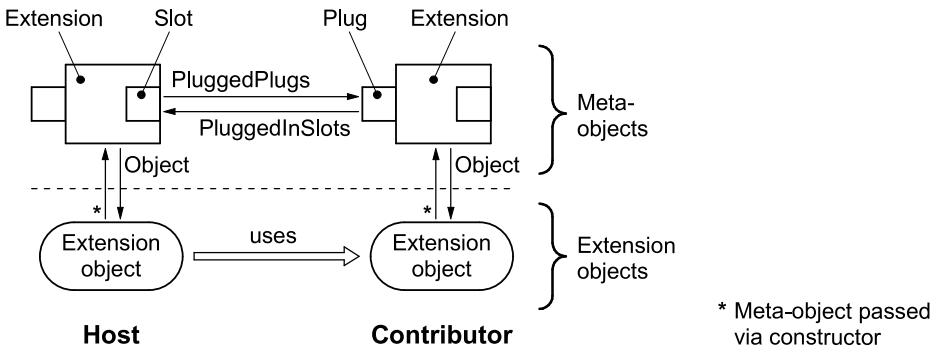


Figure 3.6: Meta-objects for instantiated extensions in the composition state

Let us show, by means of the workbench example, how the composition state can be used to create a window menu that lists the titles of all opened views (Figure 3.7). The window menu is used to set the focus to an open view and to bring it to the foreground. For each plugged view the workbench opens the view's control in a child window and it closes the child window when the view contributor is unplugged. How this is done is covered in Section 3.3.3 Composition Events later in this chapter.

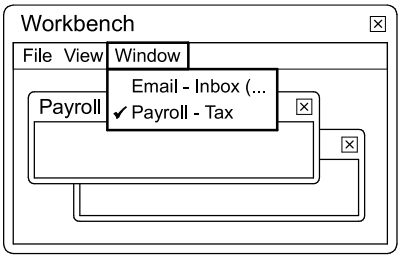


Figure 3.7: Window menu in the user interface of the workbench application

Listing 3.4 shows how the workbench creates the window menu with entries for opened views, i.e., how it creates a menu entry for each plugged view contributor. When the composer creates the *Workbench* extension, it passes the extension's meta-object to the constructor, which uses it to retrieve the meta-object of the *View* slot. When the user opens the window menu the *ShowWindowMenu* method is called. This method retrieves the plug meta-objects of the view contributors that are plugged into the *View* slot using the slot's *PluggedPlugs* property (Figure 3.6). For each retrieved plug we retrieve the extension object of the contributor. As Plux ensures that only contributors are plugged that implement the interface *IView*, which is required by the slot definition, we can safely cast the contributors extension object to *IView*. Finally, we

```
[Extension]
[Plug("Application")]
[Slot("View")]
class Workbench : IApplication {
  Slot viewSlot;

  Workbench(Extension e) { viewSlot = e.Slots["View"]; }

  void ShowWindowMenu() {
    Menu windowMenu = ...
    foreach (Plug p in viewSlot.PluggedPlugs) {
      var view = (IView) p.Extension.Object;
      String title = view.GetTitle();
      windowMenu.Add(title, p);
    }
    windowMenu.Show();
  }
}
```

Listing 3.4: Retrieving meta-objects for plugged contributors from the composition state

retrieve the dynamic title from the extension object and use it as a label for the menu entry. Additionally, we store the plug meta-object so that we can later use it to switch to the corresponding view when the user clicks the menu entry.

Besides the *plugged* relationship, the composition state also maintains the *tagged* relationship. A tag can be set between a plug and a slot, regardless of whether that plug is plugged into the slot. That means that contributors can be tagged before a host uses them, i.e., before they are plugged. This can be used to introduce contributors to a host, e.g., to make a view contributor available in the workbench menu before it is even instantiated. Contributors can also be tagged when the host already uses them, i.e., when they are already plugged. This can be used to mark one of the plugged contributors, e.g., as the current foreground view in the workbench.

Figure 3.6 shows how the contributors that are plugged in a slot can be retrieved from the composition state, either via the *PluggedPlugs* property of a slot, or via the *PluggedInSlots* property of a plug. Similarly, for every slot, the composition state keeps track of which plugs are tagged with a tag (retrievable via the slot's *Tags* property); and for every plug, the composition state keeps track of in which slots it is tagged (retrievable via the plug's *Tags* property).

In our workbench example, the *Menu* tag can be used to customize which views should be visible in the view menu, so that the menu presents available views and lets the user open them by clicking the menu entries. However, as the workbench shows only those views in its view menu which are tagged with the *Menu* tag, users can customize the view menu by tagging or untagging view contributors.

Listing 3.5 shows how the workbench uses the *Menu* tag. By setting *AutoTag="Menu"* as a property of the *View* slot, we ensure that the composer automatically tags all discovered view contributors (Section 3.3.4 Automatic Composition). When the user

```
[Extension]
[Plug("Application")]
[Slot("View", AutoTag="Menu")]
class Workbench : IApplication {
  Slot viewSlot;

  Workbench(Extension e) { viewSlot = e.Slots["View"]; }

  void ShowViewMenu {
    Menu viewMenu = ...
    foreach (Tag t in viewSlot.Tags["Menu"]) {
      String title = (String) t.Plug.Param["Title"].Value;
      viewMenu.Add(title, p);
    }
    viewMenu.Show();
  }
}
```

Listing 3.5: Retrieving meta-objects for tagged contributors from the composition state

opens the view menu, the workbench retrieves the tagged contributors from the view slot using the *Tags* property with the tag name as a filter. For each tagged contributor, we retrieve its static title from the *Title* parameter value of the plug. Finally, we add a menu entry with the title as a caption to the view menu and store also the plug meta-object so that we can later open the corresponding view when the user clicks the menu entry (see UI-bound Composition Behaviors in Section 3.3.6).

In addition to the meta-objects and relationships explained in this section, the composition state holds further composition data. These will be explained in the next Section 3.3.2 Composition Operations when the corresponding concepts are presented.

3.3.2 Composition Operations

In Plux, the composer assembles an application from extensions and stores them together with their connections in the composition state. For this purpose it uses several composition operations, which we describe in this section by means of the workbench example.

When Plux starts an application, the composition state contains only the Plux core extension, which provides an *Application* slot as a root for composition (Figure 3.8 Before). At startup, contributors with an *Application* plug are composed here.

Create

Before an extension can be composed, it must be created. To create an extension means to create the extension meta-object as well as the related slot and plug meta-objects (Figure 3.8 After) using the metadata provided by the discoverer. The fact that the workbench meta-objects in Figure 3.8 are shown with dashed lines indicates that the extension was not yet activated (see operation *Activate*), i.e., only the meta-objects exist, but the extension object was not yet instantiated.

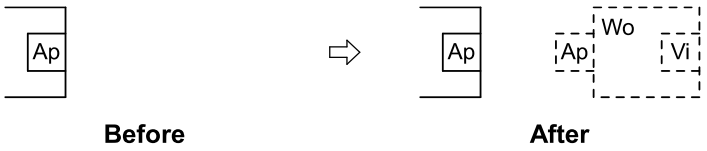


Figure 3.8: Composition state before and after the *Create* operation

Plug

Hosts can only use contributors that are plugged into them. To plug a contributor means to connect a plug with a slot in the composition state. The meta-objects of plugged contributors can be retrieved from the composition state using a slot's *PluggedPlugs* property. The host can retrieve the extension object of a plugged contributor and can call methods via the interface specified in the slot definition. Figure 3.9 shows the composition state before and after the *Plug* operation. Please note that the extension object of the workbench is still not instantiated.

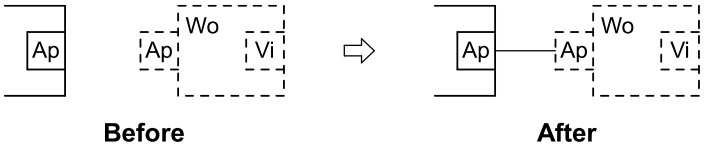


Figure 3.9: Composition state before and after the *Plug* operation

Activate

To activate an extension means to instantiate its extension object. If a host retrieves the extension object from the extension's meta-object using the *Object* property, the *Activate* operation is triggered automatically. Once a contributor has been activated, the host can call methods on the contributor's extension object via the interface of the slot to which it is connected. Whether an extension is activated or not can be retrieved from the composition state using the extension's *IsActivated* property. Figure 3.10 shows an example: the unactivated extension (Before) is drawn with dashed lines, whereas the activated extension (After) is drawn with solid lines. The fact that the slot meta-object is crossed out indicates that the slot is yet closed (see operation *Open*).

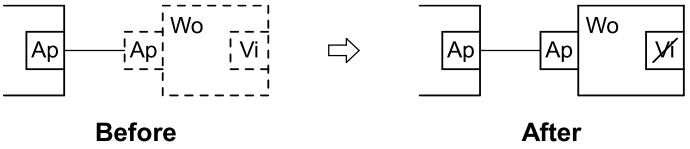


Figure 3.10: Composition state before and after the *Activate* operation

Open

Slots can be open or closed. Note that contributors can only be plugged or tagged to open slots. To open a slot means to mark it as open in the composition state. Whether a slot is open or closed can be retrieved from the composition state using the slot's *IsOpen* property. Figure 3.11 shows an example: the closed slot is struck through (Before), whereas the open slot is not (After).

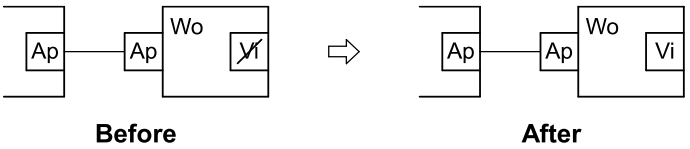


Figure 3.11: Composition state before and after the *Open* operation

Tag

To tag a slot-plug pair means to store a named relation for that pair in the composition state. A slot-plug pair can have multiple tags, namely all that were defined in the slot definition. Tags can be set even when the plug is not yet connected to the slot. The meta-object of tags can be retrieved from the composition state using a slot's *Tags* property. Whether a tag is set between a slot and a plug can be retrieved from the

composition state using the slot's *IsTagged* method. Figure 3.12a shows an example of a contributor that is tagged with *Menu* but not plugged, Figure 3.12b shows a contributor that is both plugged and tagged.

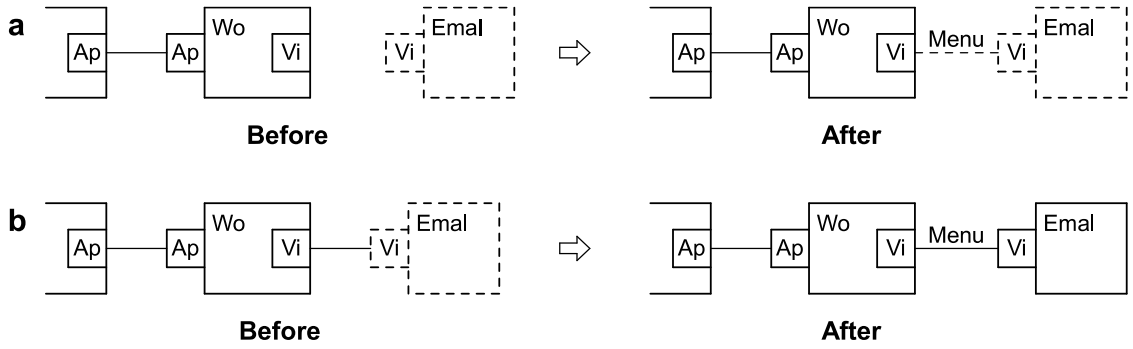


Figure 3.12: Composition state before and after the *Tag* operation

Destroy

To destroy an extension means to remove its meta-object from the composition state (Figure 3.8, Before). To destroy an extension includes its deactivation (operation *Deactivate*), if it was activated.

Unplug

To unplug means to remove a plug as a contributor for a slot in the composition state (Figure 3.9, however with Before and After swapped). Unplugged contributors are candidates for garbage collection (Section 3.3.4 Automatic Composition), i.e., the next time when the composer is idle, it checks if the unplugged contributor is connected to any other slot. If it is neither plugged nor tagged to any slot anymore, the composer destroys the contributor (operation *Destroy*), if it is still tagged to a slot but not plugged to any slot, the composer deactivates the contributor (operation *Deactivate*).

Deactivate

To deactivate an extension means that its slots are closed (operation *Close*) and that its extension object is disposed and released for garbage collection. (Figure 3.8, After). As the extension object of a deactivated extension must not be used anymore, deactivating an extension includes unplugging the extension from all hosts where it was plugged before the extension's slots are closed and the extension object is disposed.

Close

To close a slot means to mark it as closed in the composition state (Figure 3.11, however with Before and After swapped). As Plux allows plugged and tagged contributors only in open slots, to close a slot includes untagging (operation *Untag*) and unplugging (operation *Unplug*) all contributors (not shown in Figure 3.11).

Untag

To untag means to remove the named relation between a slot and a plug from the composition state (Figure 3.12, however with Before and After swapped). Similar to the *Unplug* operation, untagged contributors are candidates for garbage collection if they are not plugged to any slot anymore.

3.3.3 Composition Events

For each composition operation, the composer sends the following events to the affected components (see below): a *CanCompose* event asks the receiver whether the operation can be performed, if one receiver denies the request, the composer cancels the operation; a *Composing* event notifies that the operation is about to be performed so that receivers can prepare for the upcoming change; a *Composed* event notifies that the operation has been completed so that receivers can react to the change. Although composition events are primarily useful for hosts, other components, such as contributors or system tools, can also receive the events. Hosts can register for events on their extension and slot meta-objects, contributors can register for events on their plug meta-objects, and system tools can register for global events on the composer.

For each composition operation, there is a specific variant of the *CanCompose*, *Composing*, and *Composed* events, e.g., *CanPlug*, *Plugging*, and *Plugged* for the *Plug* operation. Figure 3.13 lists the events for all composition operations and shows where receivers can register for them.

Composition Event

Composition Operation	CanCompose		Composer	Extension	Slot	Plug	Composing		Composer	Extension	Slot	Plug	Composed		Composer	Extension	Slot	Plug
Create	**CanCreate**		×				**Creating**		×				**Created**		×			
Activate	**CanActivate**		×	×			**Activating**		×	×			**Acitvated**		×	×		
Open	**CanOpen**		×		×		**Opening**		×		×		**Opened**		×		×	
Tag	**CanTag**		×		×	×	**Tagging**		×		×	×	**Tagged**		×		×	×
Plug	**CanPlug**		×		×	×	**Plugging**		×		×	×	**Plugged**		×		×	×
Destroy	**CanDestroy**		×				**Destroying**		×				**Destroyed**		×			
Deactivate	**CanDeactivate**		×	×			**Deactivating**		×	×			**Deactivated**		×	×		
Close	**CanClose**		×		×		**Closing**		×		×		**Closed**		×		×	
Untag	**CanUntag**		×		×	×	**Untagging**		×		×	×	**Untagged**		×		×	×
Unplug	**CanUnplug**		×		×	×	**Unplugging**		×		×	×	**Unplugged**		×		×	×

Source **Source** **Source**

Figure 3.13: Composition events for the Plux composition operations

In the workbench example, we use the composition events of the plug and unplug operations to open and close views. However, before we show how this is done, we modify our workbench, in order to make the arrangement of views customizable. We

factor out the view arrangement logic from the *Workbench* extension and introduce an additional slot for containers instead. The views are now arranged by a contributor for the *Container* slot. Figure 3.14 shows two examples for container contributors: the *MdiContainer* contains the original logic, which arranges the views as child windows; the *TabContainer* is an alternative which arranges the views on tabs.

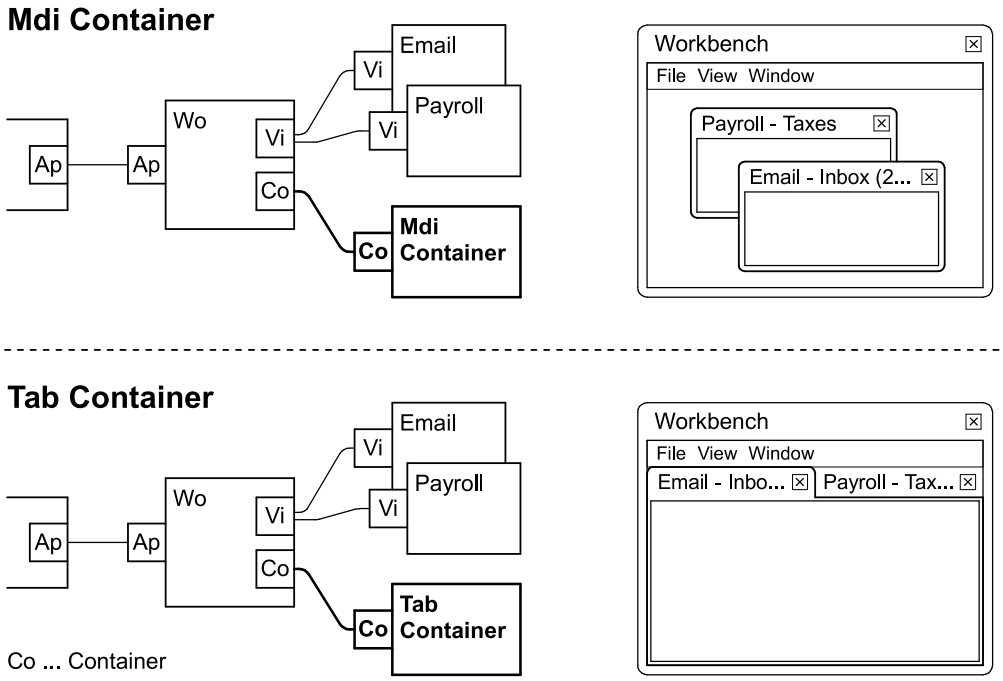


Figure 3.14: Customizable view arrangement in the workbench example using containers

Listing 3.6 shows the modified version of the workbench with the *Container* slot and how it handles the composition events for the *View* slot in order to open and close views when they are plugged and unplugged. In order to do so, it registers event handler methods for the *CanPlug*, the *Plugged*, and the *Unplugging* events in the slot attribute for the *View* slot.

Please note that in contrast to previous implementations of the workbench, the *Workbench* class inherits from the base class *ExtensionBase* now and thus does not need to store references to the meta-objects of its slots itself, but retrieves them via the *Slots* property in the base class.

As the workbench can only arrange views if a container is plugged, it registers the *IsContainerPlugged* method as an event handler for the *View* slot's *CanPlug* event. The event handler denies plug operations for views if no container is present and writes a log message. After a container has become available, the workbench is ready for views. As soon a view is plugged the workbench opens it. For this, it registers the *OpenView* method as an event handler for the *View* slot's *Plugged* event. The event handler retrieves the plugged container via the *Container* slot and calls the container's *Open*

```
[Extension]
[Plug("Application")]
[Slot("View", AutoTag="Menu", CanPlug="IsContainerPlugged",
      Plugged="OpenView", Unplugging="CloseView")]
[Slot("Container")]
class Workbench : ExtensionBase, IApplication {

  bool IsContainerPlugged(CompositionEventArgs args) {
    if (Slots["Container"].PluggedPlugs.Count == 0) {
      args.Logger.Write(
          "View denied because no container is plugged.");
      return false;
    }
    return true;
  }

  IContainer Container {
    get { return (IContainer)
      Slots["Container"].PluggedPlugs[0].Extension.Object; }
  }

  void OpenView(CompositionEventArgs args) {
    Container.Open((IView) args.Plug.Extension.Object);
  }

  void CloseView(CompositionEventArgs args) {
    Container.Close((IView) args.Plug.Extension.Object);
  }
}
```

Listing 3.6: Handling *CanPlug*, *Plugged*, and *Unplugging* composition events

method with the currently plugged view as an argument. Just before a view is unplugged, the workbench closes it. For this, it registers the *CloseView* method as an event handler for the *View* slot's *Unplugging* event. When invoked, the event handler closes the view in the container.

In the methods *OpenView* and *CloseView*, we retrieve the container without even checking if a container is plugged, because the *CanPlug* event handler ensures that views can only be plugged once a container is plugged. Please note, that the implementation in Listing 3.6 is not sufficient for all situations, e.g., the workbench does not handle the situation where the container is unplugged while views are plugged. A solution for this problem is shown in a further improved implementation of the workbench in Section 3.3.5 Programmatic Composition.

3.3.4 Automatic Composition

Automatic composition is the process performed by the composer where new extensions are created and connected (see Composition Process below), or extensions are disconnected and destroyed (see Decomposition Process below).

Composition Process

The composition process defines how extensions are composed to a program, i.e., it creates extensions and connects them to other extensions. The composition process is divided into composition sequences. A sequence comprises the composition operations that are necessary to compose one host with all contributors that are available for the slots of this host, i.e., each composition sequence makes one extension ready for use. In a composition sequence, the composer performs the following composition operations in the given order: (1) it activates the host, (2) opens the first slot of the host, (3) creates the contributors for this slot, (4) tags them with the specified tags, and (5) plugs them into the slot. If the host has multiple slots, the composer composes them one after another. If it has no slots, the sequence is completed after activating the host.

Figure 3.15 shows the composition sequence in which the composer composes the *Workbench* extension, i.e., the *Workbench* extension is the host and the *Email* extension is the contributor. In this sequence the composer activates the *Workbench* host, opens its *View* slot, creates the *Email* contributor, tags it with the *Menu* tag, and finally plugs it into the *View* slot. Now the *Workbench* is completely composed and ready for use.

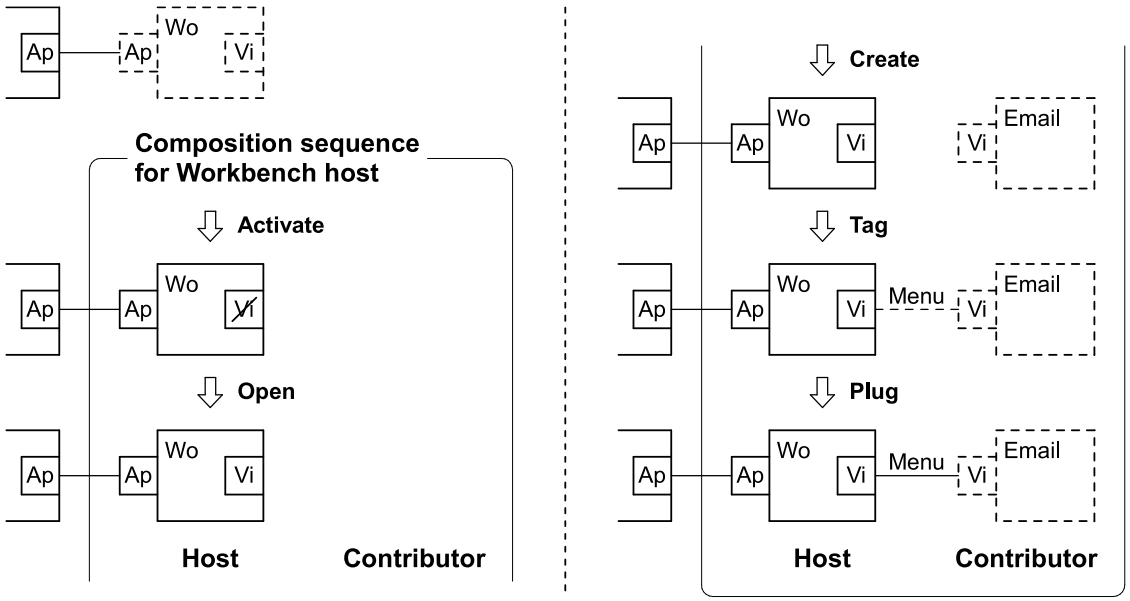


Figure 3.15: Composition sequence comprising the composition operations that compose a host with a contributor

In automatic composition, the order of the composition operations in a composition sequence cannot be changed. However, which composition operations should be performed automatically and which should not can be configured globally on the composer or individually for each slot and plug. Thus, depending on the configuration of the composer or of a meta-object, a composition sequence performs or skips a composition operation. The default is that automatic composition is enabled for all composition operations, except for the *Tag* operation. To enable automatic composition

for the *Tag* operation, the *AutoTag* property of a slot needs to be set, either in its declaring attribute or in its meta-object. As the *View* slot of the *Workbench* specifies *AutoTag="Menu"* in its slot attribute (Listing 3.6), the *Menu* tag is set during automatic composition. To disable automatic composition for a composition operation, the corresponding composition property of the composer or of a meta-object needs to be disabled. For example, setting *AutoOpen=false* for a slot, disables the *Open* operation in a composition sequence for this slot. Of course, as contributors can be tagged and plugged only into open slots, disabling the *Open* operation for a slot causes the operations *Tag* and *Plug* to be skipped in the composition sequence, too. However, if automatic composition is enabled for the *Tag* and *Plug* operations, the composer automatically performs them as soon as the slot gets opened later (see 3.3.5 Programmatic Composition).

Figure 3.16 shows the composition properties that can be set to enable or disable automatic composition for a certain composition operation, either globally for all slots and plugs or individually for specific slots and plugs. Automatic composition for the operations *Create* and *Activate* cannot be disabled.

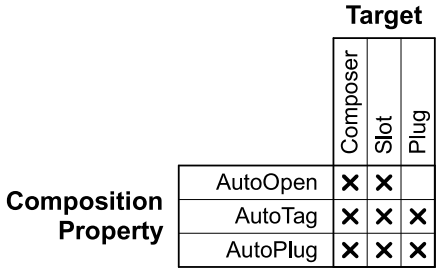


Figure 3.16: Composition properties to enable or disable automatic composition for specific composition operations

After a composition sequence is completed, the composer is idle until a new composition sequence is triggered. In automatic composition, a composition sequence is triggered either when a host retrieves the extension object of a not yet activated contributor (see Host-triggered Composition) or when the discoverer adds a new extension (see Discoverer-triggered Composition).

Host-triggered Composition

In host-triggered composition, the composer starts a composition sequence, when a host tries to retrieve the extension object of a not yet activated contributor. As the new composition sequence is started for the accessed contributor, this contributor gets composed just in time before the extension object is returned to the host. Please note, that in this context, the accessed contributor is now the host to be composed, i.e., it is the extension that gets activated and whose slots get filled with further contributors. With host-triggered composition, Plux ensures that a contributor always is activated and its slots are filled when it is accessed by a host.

When a Plux application is started, the composition starts with the composition sequence for the Plux core. The Plux core listens to the *Plugged* event of its *Application* slot and retrieves the extension object of the contributor when it is plugged. Thereby the Plux core triggers a composition sequence for the newly plugged contributor. Listing 3.7 shows the event handler for the *Plugged* event at the *Application* slot of the Plux core retrieving the extension object of its contributor. When it calls the contributor's *Start* method, the contributor is already composed.

```
[Extension]
[Slot("Application", Plugged="StartApplication")]
class Core {
  void StartApplication(CompositionEventArgs args) }
    IApplication app = (IApplication) args.Plug.Extension.Object;
    app.Start();
  }
}
```

Listing 3.7: Host retrieving the extension object of its contributor in the *Plugged* event handler

Figure 3.17 shows the composition process for the workbench example. In the bootstrap composition sequence 1, the Plux core is the host to be composed. The composer activates the Plux core and fills the *Application* slot. When the Plux core retrieves the extension object of the *Workbench* contributor in response to the *Plugged* event in sequence 1 (i.e., while composition sequence 1 has not yet completed), sequence 2 is triggered as a subsequence of sequence 1. As sequence 2 is a subsequence of sequence 1, sequence 2 completes before sequence 1.

In sequence 2, the *Workbench* is the host to be composed and the *Email* view is the contributor. When the *Workbench* retrieves the extension object of the *Email* view in its *Plugged* event handler, it triggers sequence 3 as a subsequence of sequence 2. In sequence 3, the *Email* view is the host. As the *Email* view has no slots, sequence 3 has no contributors to compose and is completed after the *Email* view has been activated. The sequences 2 and 1 are also completed because they have no further composition operations to be performed. This concludes the composition process for the workbench application and the workbench is ready for use.

The composition of a host may trigger the composition of its contributors recursively. Thus, whenever a host retrieves the extension object of one of its contributors, not only this contributor itself is guaranteed to be already composed, but also the contributors of this contributor if their extension objects were accessed.

As the composer only starts a composition sequence for a contributor when the contributor's extension object is retrieved, the composer only composes a minimal set of extensions that are in use, thus guaranteeing fast startup times of Plux-based applications. For example, if the workbench would not retrieve the views' extension objects immediately in its *Plugged* event handler, the composer would not compose

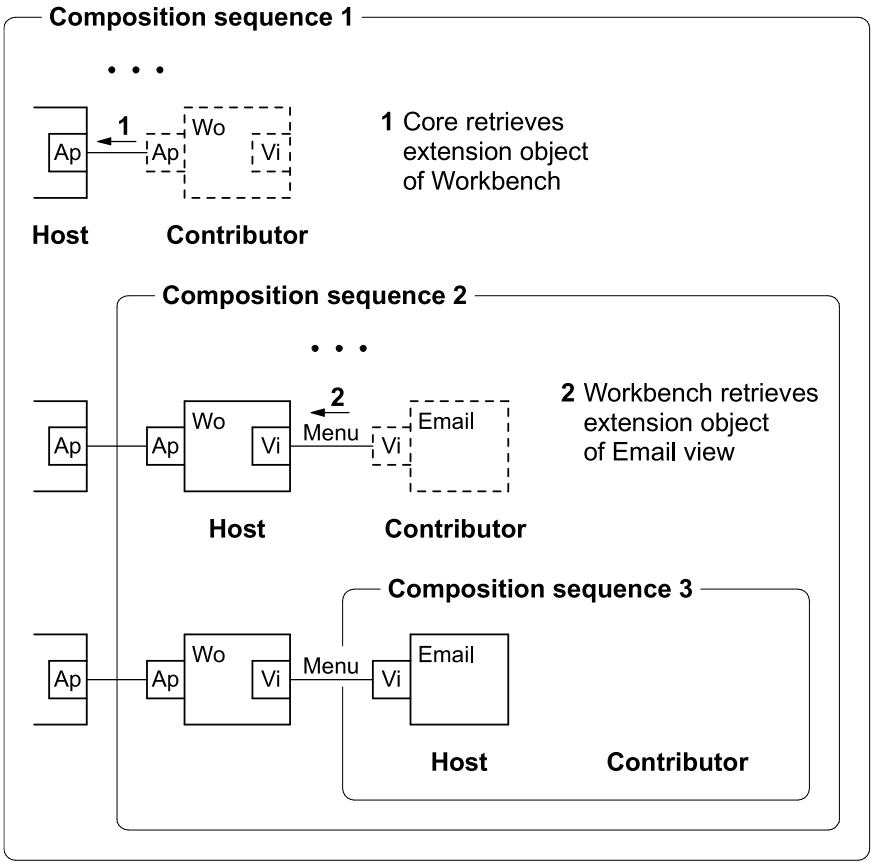


Figure 3.17: Composition sequences triggered as subsequences by hosts that retrieve the extension objects of their contributors

view extensions until they are used. In Section 3.3.6 Behavior-guided Composition, we will modify the workbench so that it does not open each view immediately, but only when the user clicks an item from the view menu.

Discoverer-triggered Composition

Composition sequences are also triggered when the discoverer adds new extensions. For each newly added extension, the composer searches the composition state for matching slots and composes the new contributors, i.e., it creates, tags and plugs them. However, composition sequences that are triggered by the discoverer do not include the *Activate* and the *Open* operations on the host because the hosts were already composed in prior composition sequences. In Figure 3.18 on the next page, the discoverer adds a new *Payroll* view as a contributor for a *View* slot. As the composer finds the *Workbench* as a matching host for the *Payroll* view, it starts composition sequence 4, which creates the *Payroll* view, tags it with the *Menu* tag, and plugs it into the *View* slot of the *Workbench* extension (not shown).

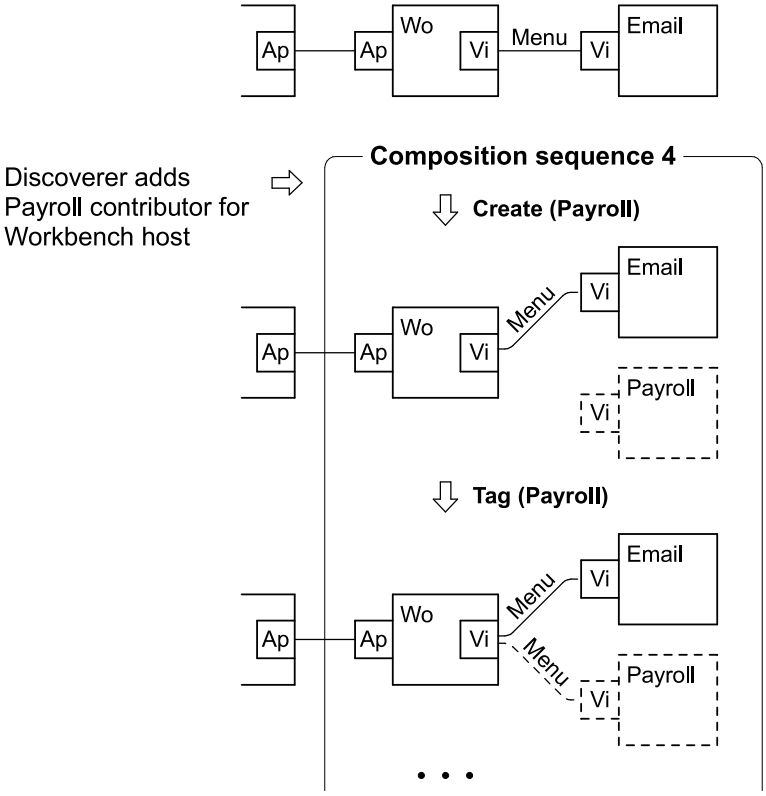


Figure 3.18: Composition sequence triggered by a discoverer that adds a new extension

Composition Sequences with Multiple Contributors

The composition sequences, shown so far, covered only scenarios with a single contributor that was composed per composition sequence. However, if multiple contributors are available for a slot, a composition sequence composes all available contributors. Such a composition sequence is performed as follows: the composer activates the host and opens the slot of the host. Next the composer creates and tags all contributors, before it plugs the contributors. The order in which contributors are composed is not specified.

Figure 3.19 shows an updated version of composition sequence 2 from Figure 3.17 on page 58, this time, however, with multiple contributors available at the same time: the *Email* view and the *Payroll* view. The composer activates the *Workbench* and opens its *View* slot. After that, the composer creates and tags the contributors *Email* and *Payroll*. Next, the composer plugs the first contributor, i.e., the *Email* view. When the *Workbench* retrieves the extension object of the *Email* view, it triggers composition sequence 3, where the *Email* view is activated. As the *Email* view does not have slots, sequence 3 is completed after that. Finally, the *Payroll* view is plugged and activated in the same way.

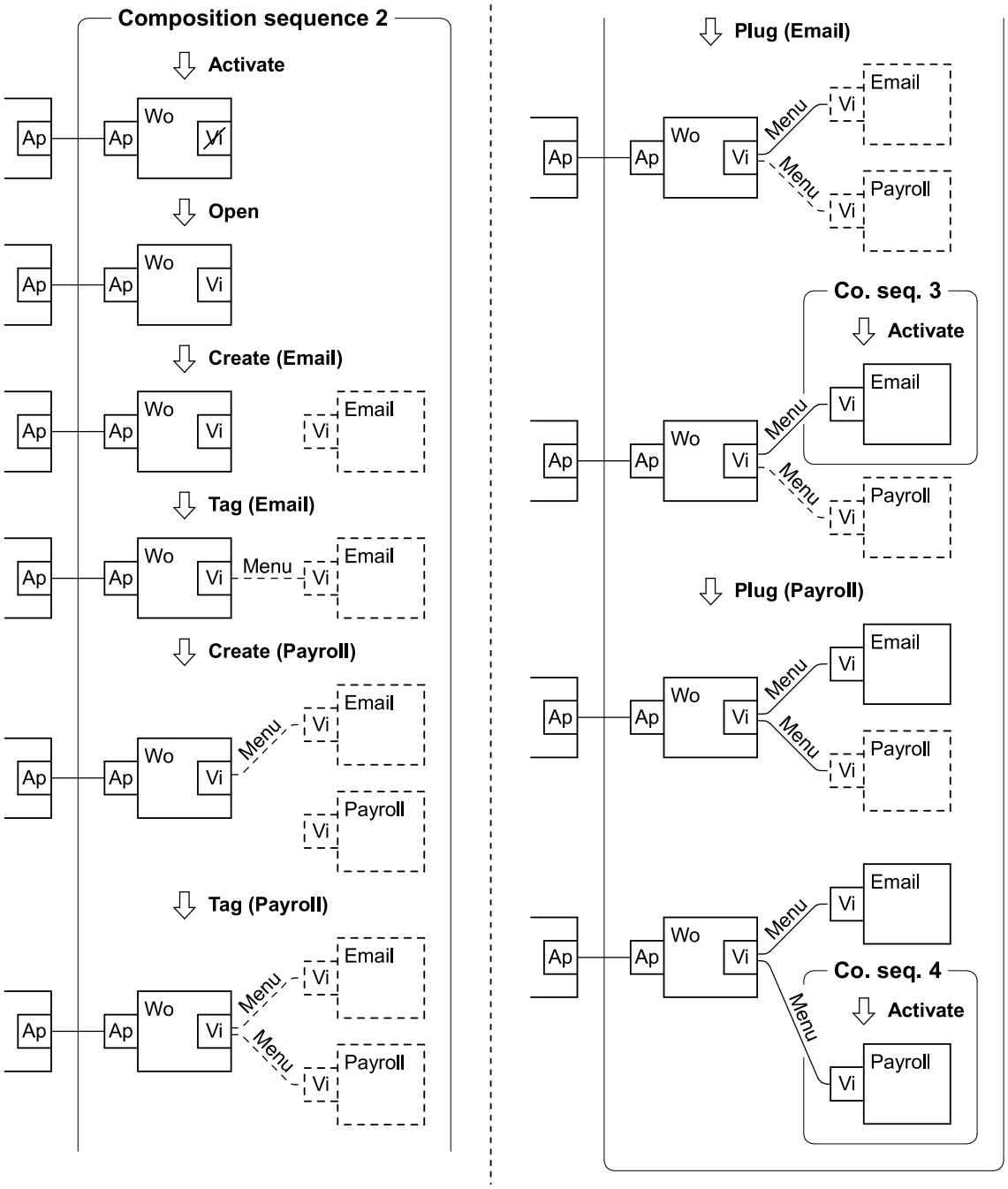


Figure 3.19: Composition sequence comprising the composition operations that compose a host with multiple contributors

Non-shared versus Shared Contributors

In automatic composition, the composer distinguishes between non-shared and shared contributor instances. A *non-shared* instance is connected to only a single host, whereas a *shared* instance is connected to multiple hosts and thus is shared among them. Hosts

can specify in their metadata whether they want a non-shared or a shared contributor to be connected, the default being a non-shared contributor.

Let us extend the workbench example, so that the *Email* view and the *Payroll* view share a *DataModel* contributor, e.g., a common address book. Figure 3.20 shows the difference between a non-shared and a shared contributor using the workbench example.

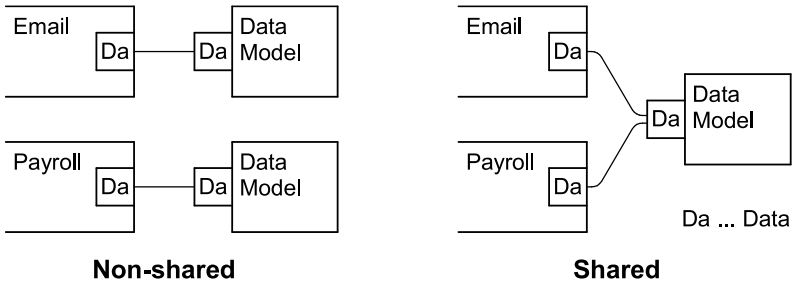


Figure 3.20: Multiple hosts use separate instances of a contributor (non-shared), or use a common instance of a contributor (shared)

As we want to use the same data both in *Email* and in *Payroll*, we choose the shared data model. Listing 3.8 shows the metadata of the *Email* view, which specify that it requests a shared contributor in its *Data* slot.

```
[Extension]
[Plug("View")]
[Param("Title", "Email")]
[Slot("Data", Shared=true)]
class Email : IView {
  ...
}
```

Listing 3.8: Metadata of a host with a slot for shared contributors

If a host specifies that it requests a shared contributor, during composition the composer either reuses a specific shared instance of the contributor, which was already created in a prior composition sequence, or the composer creates a new specific shared contributor instance, which is reused later for other hosts that also request a shared contributor.

Decomposition Process

The decomposition process defines how extensions are decomposed from a program, i.e., how the composer disconnects and destroys them. Similar to the composition process, the decomposition process is divided into sequences. A decomposition sequence comprises the composition operations that are necessary to decompose a host. In a decomposition sequence the composer performs the following composition operations in the given order for every slot: it untags all contributors from the slot,

unplugs them from the slot, and closes the slot. Finally, it deactivates and destroys the host. Figure 3.21 shows the decomposition of the *Workbench* extension in decomposition sequence 2. In this sequence the composer removes the *Menu* tag, unplugs the *Email* view, closes the *View* slot, and finally deactivates and destroys the *Workbench* extension. Decomposition is triggered by the garbage collector as explained in the next section.

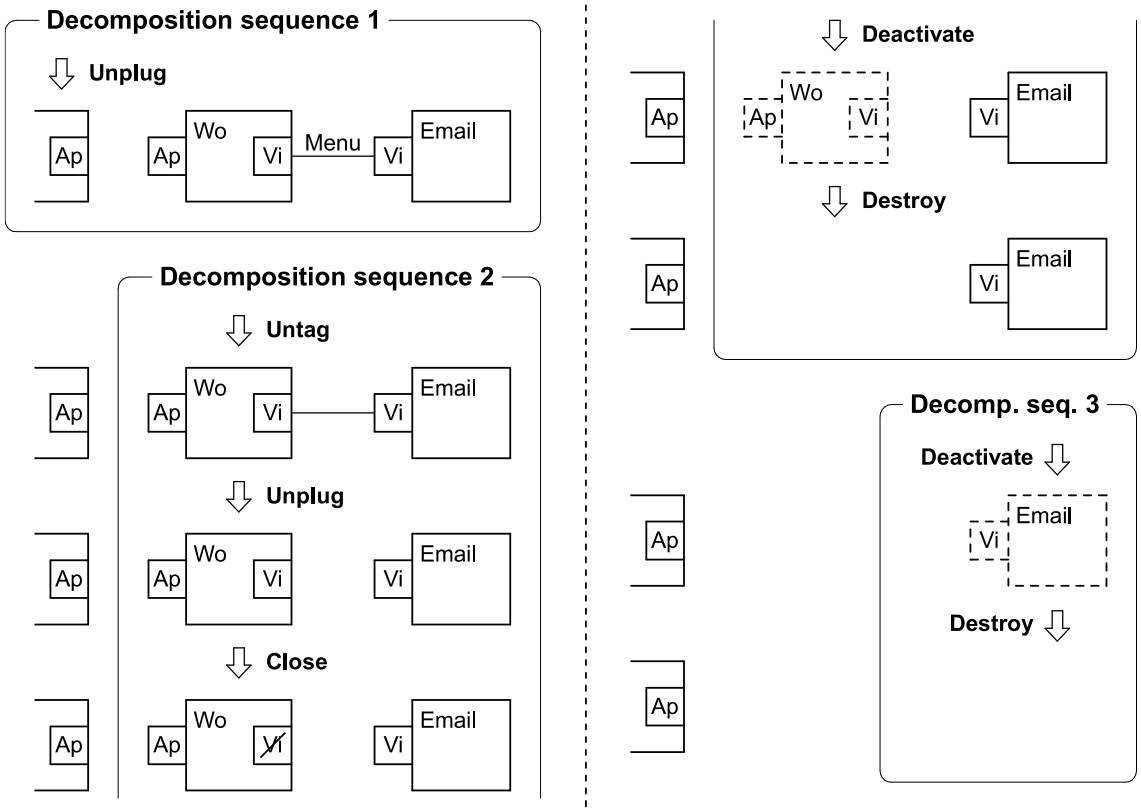


Figure 3.21: Decomposition sequences triggered by the Plux garbage collector after a host was unplugged

Garbage Collection

Plux provides a special garbage collector that destroys extensions that are not used anymore. Contributors those are untagged and unplugged during a decomposition sequence become candidates for garbage collection. Every time when Plux becomes idle, it starts the garbage collector. Extensions that are neither tagged nor plugged in any slots cannot be used anymore and thus the garbage collector triggers a decomposition sequence for them. Extensions that are not plugged but still tagged to a slot are not used at the moment and thus get deactivated, whereby all contributors of the deactivated extension get disconnected (see operation *Deactivate* in Section 3.3.2 Composition Operations) and therefore become new candidates for garbage collection.

In contrast to composition sequences, decomposition sequences are not nested, i.e., a contributor is not immediately decomposed after it was unplugged; instead, decomposition starts when Plux is idle again, thus an unplugged contributor can be moved from one slot to another without being garbage-collected instantly. In other words, it is possible to unplug the contributor in one decomposition sequence and to plug it in another composition sequence. If decomposition sequences were nested, the contributor would already be destroyed or deactivated before it could be plugged into the new slot.

In Figure 3.21 on page 62 the *Workbench* is unplugged in decomposition sequence 1 and becomes a candidate for garbage collection. When Plux becomes idle, it triggers decomposition sequence 2 for the *Workbench* because it is neither tagged nor plugged in any slot. The same happens with the *Email* view. It is untagged and unplugged in sequence 2 and decomposed in sequence 3 the next time Plux becomes idle. Finally, the *Workbench* and its contributors are removed by the Plux garbage collector and the *Application* slot of the *Core* is empty. This decomposition process ensures automatic decomposition, i.e., when a host is disconnected, all its contributors as well as their contributors are disconnected and destroyed recursively as well if they are not connected to any other host.

3.3.5 Programmatic Composition

Automatic composition is sufficient for many situations, however in some situations developers need more control over which contributors should be connected to a host and which should not. For this purpose, Plux allows developers to partially disable the automatic composition process by configuring the composer, so that it omits certain composition operations. For example, the composer might be configured to open slots and to tag contributors automatically, but to omit automatic plugging of contributors for a certain slot. This configuration can be done for individual slots, for individual plugs, or for all slots and plugs in the composition state. As a substitute for the disabled automatic composition operations, developers can call composition operations programmatically using the composer's API.

Programmatic composition can be combined with automatic composition. For example, if a host wants to make sure that its slots are filled in a certain order, it can use automatic composition to create and plug the contributors, but use programmatic composition to control when the slots are opened. For doing so, the host disables the automatic *Open* composition operation for the slots that it wants to control and calls the *Open* composition operation programmatically for these slots when the time has come.

Figure 3.22 shows an example for such a scenario with the *Workbench* extension that has two slots. The first slot is for views; everything that plugs here is displayed within the workbench. The other slot is for containers; the contributor that plugs here controls how the views are arranged within the workbench. There is some kind of relationship between the two slots. Without a container, the workbench cannot arrange the views.

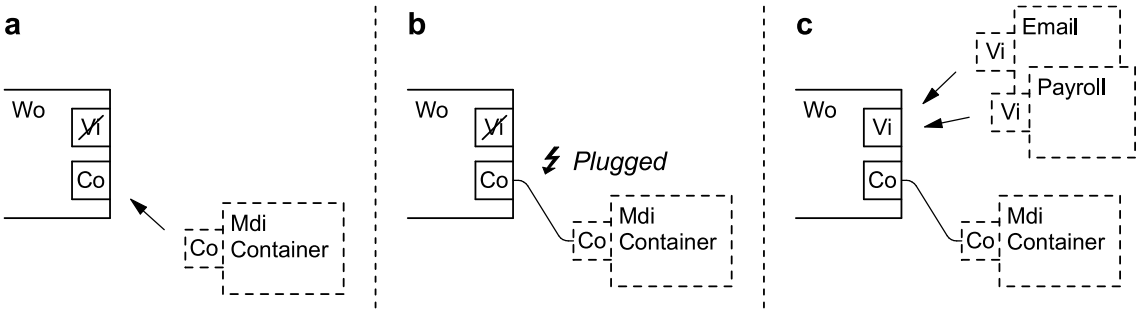


Figure 3.22: Relationship between two slots that have to be filled in a certain order

Therefore in (a), the *View* slot has to be kept closed until a container is plugged. When in (b) a container is plugged, in (c) the *View* slot should be opened and filled with view contributors.

The scenario of Figure 3.22 cannot be achieved with automatic composition because automatic composition would open and fill the *View* slot immediately, possibly before the *Container* slot. There would be no guarantee that the *Container* slot is filled before the *View* slot. To achieve the desired composition order, we use programmatic composition to control the composition process as shown in Figure 3.23

The *Workbench* disables the *Open* operation for its View slot, thus the automatic composition does not open this slot after activating the *Workbench*. Then the automatic

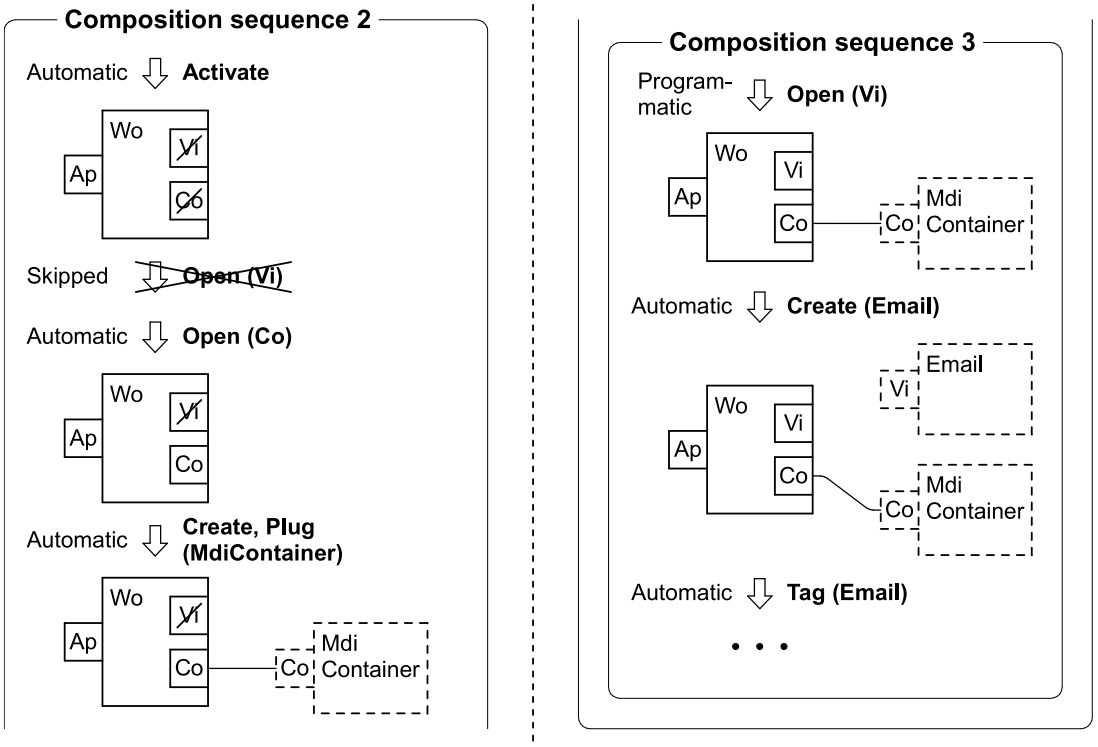


Figure 3.23: Controlling the composition process by combining automatic and programmatic composition

composition opens the *Container* slot and fills it with the *MdiContainer*. In reaction to the *Plugged* event of the *Container* slot the *Workbench* can now call the composition operation *Open* for the *View* slot programmatically. This triggers a new composition sequence, whereby the automatic composition fills the *View* slot with the *Email* contributor. This composition process composes the *Workbench* in the desired order, as views are only composed after a container was composed.

Listing 3.9 shows the implementation of the modified *Workbench* host. The assignment *AutoOpen=false* in the *View* slot's attribute disables the *Open* composition operation in automatic composition. Instead, the workbench performs this operation now programmatically in the *Plugged* event handler of the *Container* slot: when a container is plugged, the *Plugged* event handler calls the *View* slot's *Open* method. Vice versa, just before a container is unplugged, the *Unplugging* event handler closes the *View* slot by calling the slot's *Close* method.

```
[Extension]
[Plug("Application")]
[Slot("View", AutoOpen=false, ...)]
[Slot("Container", Plugged="ContainerPlugged",
    Unplugging="ContainerUnplugging")]
class Workbench : ExtensionBase, IApplication {
  ...
  void ContainerPlugged(PlugEventArgs args) {
    Slots["View"].Open();
  }
  void ContainerUnplugging(PlugEventArgs args) {
    Slots["View"].Close();
  }
}
```

Listing 3.9: Implementation of a host that calls composition operations programmatically

Figure 3.24 shows the composition operations that can be performed programmatically for extensions, slots, and plugs. The operations *Tag*, *Plug*, *Untag*, and *Unplug* have a parameter for the opposite slot or plug, e.g., if the *Plug* operation is applied on a slot, it

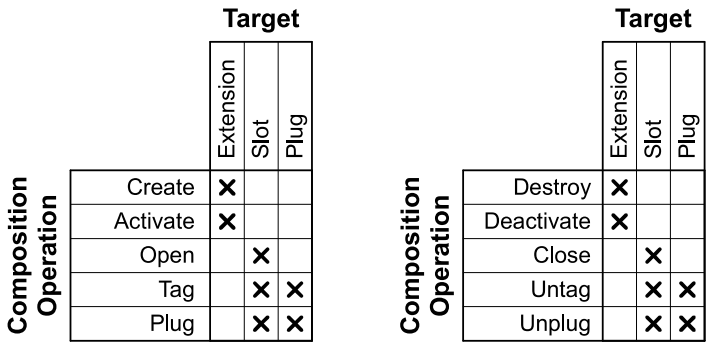


Figure 3.24: Composition operations for extensions, slots, and plugs

gets the opposite plug as an argument; if it is applied on a plug, it gets the opposite slot as an argument. The *Tag* and *Untag* operations also get the name of the applied tag as an argument.

3.3.6 Behavior-guided Composition

Programmatic composition allows developers to write custom composition logic to control the composition process. However, this composition logic can bloat the implementation of an extension. Furthermore, it is hidden inside the extensions and thus cannot be reused in other extensions. Extensive use of programmatic composition can lead to the following problems: it duplicates code because common composition logic has to be repeatedly implemented in many extensions and it is error-prone because it requires detailed understanding of the composition process.

To avoid programmatic composition, Plux supports *composition behaviors*, which are reusable composition logic that can be applied declaratively to individual slots or globally to all slots in the composition state. A composition behavior targets a specific composition problem, for example, the requirement that an extension is automatically unplugged from a slot when some other extension is plugged there. In order to achieve this, the composition behavior reacts to the composition events of a slot and applies its composition logic by performing or blocking composition operations when the composition events occur.

The composition logic of a composition behavior can be implemented in three different ways. In s*elf-contained composition behaviors* the composition logic is implemented in the behavior itself, in *rule-based composition behaviors* the composition logic is extracted into a generic composition rule, which can be reused by different composition behaviors, and in *UI-bound composition behaviors* the composition logic is bound to the application's user interface, i.e., the composition logic is provided by the user who interacts with the application. The following sections describe the different types of composition behaviors in detail.

Self-contained Composition Behaviors

A self-contained composition behavior is a class that implements the composition logic necessary to achieve a desired composition result. It reacts to composition events, retrieves the composition state, and performs or blocks composition operations.

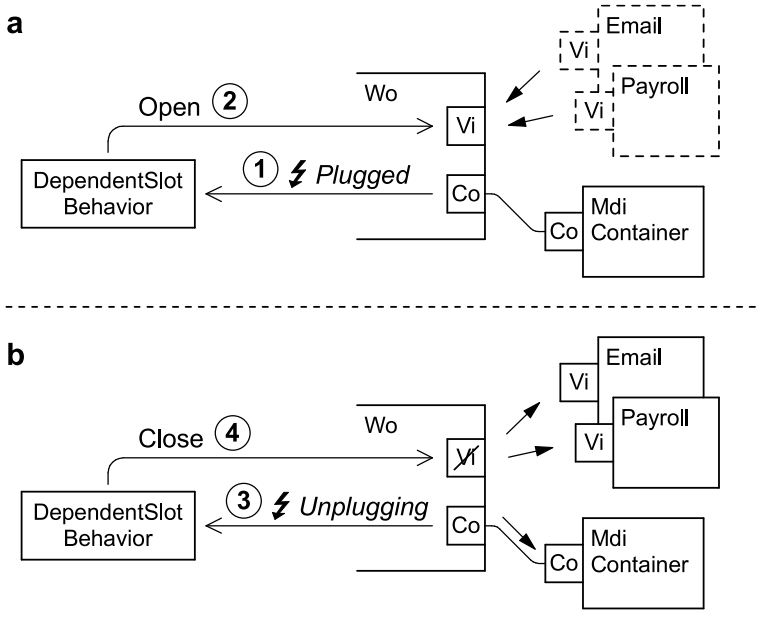
In the workbench example from Listing 3.9 on page 65 the host used programmatic composition to make sure that the view slot was only opened after the container slot was filled. Now, in order to separate this composition logic from the host implementation, we extract it into a self-contained composition behavior. As the *View* slot is depending on the *Container* slot's composition state, we call the behavior *DependentSlotBehavior*.

Listing 3.10 shows how the *DependentSlotBehavior* is attached to the *Container* slot of the *Workbench* extension. The *DependentSlotBehavior* uses two slots: it reacts to events on the *Container* slot and performs operations on the *View* slot. As the behavior is attached to the *Container* slot, it can retrieve this slot via the behavior base class (see property *BehaviorSlot* in Listing 3.11). The dependent *View* slot is passed as an argument to the behavior's constructor. Please note how composition behaviors simplify the reuse of composition logic. Composition logic that is extracted to a behavior can be used just by putting this single line of code into the constructor of a host extension.

```
[Extension]
[Plug("Application")]
[Slot("View")]
[Slot("Container")]
class Workbench : ExtensionBase, IApplication {
  Workbench() {
    Slots["Container"].Behaviors.Add(
        new DependentSlotBehavior(Slots["View"]));
  }
}
```

Listing 3.10: Attaching a composition behavior to a slot

Figure 3.25 shows how the *DependentSlotBehavior* works: (1) when it receives a *Plugged* event from the *Container* slot, (2) it performs the *Open* operation on the *View* slot; vice versa, (3) when it receives an *Unplugging* event from the *Container* slot, (4) it performs the *Close* operation on the *View* slot. As the *View* slot must not be opened without a container, e.g., if other extensions try to open it with programmatic composition, the *DependentSlotBehavior* registers an event handler for the *CanOpen* event of the View slot to allow or block the *Open* composition operation as follows: (5) when the behavior receives a *CanOpen* event from the *View* slot, (6) it checks the composition state, and blocks the *Open* operation if no contributor is plugged in the *Container* slot.



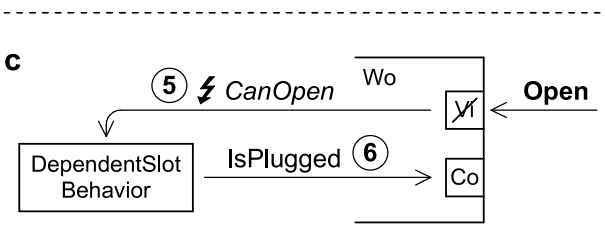


Figure 3.25: Composition behavior performing and blocking composition operations depending on composition events and the composition state

Listing 3.11 shows the implementation of the *DependentSlotBehavior*. Every composition behavior is derived from the common base class *CompositionBehavior* and inherits the property *BehaviorSlot*, which is a reference to the slot to which the behavior is attached. The constructor of the *DependentSlotBehavior* sets the dependent slot that should be controlled by the behavior. When the behavior is attached to a slot, Plux calls the *Bind* method, which registers the *CanOpenDependentSlot* method for the *CanOpen* event of the dependent slot. When the behavior is detached from a slot, Plux calls the *Unbind* method.

```
class DependentSlotBehavior : CompositionBehavior {
  Slot dependentSlot;

  DependentSlotBehavior(Slot dependentSlot) {
    this.dependentSlot = dependentSlot;
  }

  override void Bind() {
    dependentSlot.CanOpen += CanOpenDependentSlot;
  }
  override void Unbind() {
    dependentSlot.CanOpen -= CanOpenDependentSlot;
  }

  boolean CanOpenDependentSlot(SlotEventArgs args) {
    return BehaviorSlot.PluggedPlugs.Count > 0;
  }

  override void OnPlugged(PlugEventArgs args) {
    if (BehaviorSlot.PluggedPlugs.Count == 1) { dependentSlot.Open(); }
  }
  override void OnUnplugging(PlugEventArgs args) {
    if (BehaviorSlot.PluggedPlugs.Count == 1) { dependentSlot.Close(); }
  }
}
```

Listing 3.11: Implementation of a self-contained composition behavior

The *CanOpenDependentSlot* method checks the composition state of the behavior slot and blocks the *Open* operation if no contributor is plugged. In order to react to events of the behavior slot, a behavior overrides the relevant event handler methods. The *DependentSlotBehavior* overrides the *OnPlugged* and *OnUnplugging* methods: in *OnPlugged* it opens the dependent slot when the first contributor is plugged into the

behavior slot; in *OnUnplugging* it closes the dependent slot when the last contributor is about to be unplugged from the behavior slot. This ensures that the dependent slot is opened only when at least one contributor is plugged into the behavior slot.

Listing 3.12 shows the base class for composition behaviors with the event handlers that are called in reaction to the composition events. The listing shows only the *Bind* and the *Unbind* methods as well as the handlers for the events that are raised during the composition operations *Activate*, *Open*, and *Unplug*. Similar handlers exist for all other composition events (see Figure 3.13 on page 52). In the base class, the *CanCompose* event handlers (i.e., *CanActivate*, *CanOpen*, or *CanUnplug*) return *true* by default. All other event handlers are implemented empty, so that subclasses only need to override those methods that are relevant to them.

```
abstract class CompositionBehavior {
  Slot behaviorSlot;
  ...
  void Bind(Slot slot) {
    behaviorSlot = slot;
    Bind();
  }
  void Unbind(Slot slot) { /* not shown */ }

  virtual void Bind() { }
  virtual void Unbind() { }

  virtual boolean CanActivate(ExtensionEventArgs args) { return true; }
  virtual void Activating(ExtensionEventArgs args) { }
  virtual void Activated(ExtensionEventArgs args) { }

  virtual boolean CanOpen(SlotEventArgs args) { return true; }
  virtual void Opening(SlotEventArgs args) { }
  virtual void Opened(SlotEventArgs args) { }

  ...

  virtual boolean CanUnplug(PlugEventArgs args) { return true; }
  virtual void Unplugging(PlugEventArgs args) { }
  virtual void Unplugged(PlugEventArgs args) { }
}
```

Listing 3.12: Composition event handlers in the base class for composition behaviors

Rule-based Composition Behaviors

Many composition behaviors only apply to a single composition operation (e.g., *Plug*), i.e., they only react to composition events of one specific composition operation, they only retrieve the composition state composed by this operation, and they perform or block only this operation (although their composition logic usually differs from the composition logic of other behaviors). On the other hand, many behaviors implement the same composition logic, but apply to different composition operations. In order to increase reusability, rule-based composition behaviors extract the implementation of

the composition logic from a composition behavior into a reusable generic composition rule.

The *DependentSlotBehavior*, for example, only depends on whether the behavior slot is filled and whether *Plugged* or *Unplugging* events have occurred. The actual composition logic controlling the target slot, however, can be extracted into a separate rule, so that both, the composition rule and the composition behavior, can be reused with other behaviors and rules (see examples below). As the *DependentSlotBehavior* concerns the *Plug* operation of the behavior slot, and the composition logic opens and closes the target slot, we separate this behavior into a rule-based *PlugBehavior* and a generic *OpenSlotRule*. In doing so, the *OpenSlotRule* can now also be combined with another rule-based behavior, e.g., with a *TagBehavior*, to open and close a slot depending on whether a contributor is tagged in the behavior slot. Vice versa, the *PlugBehavior* can also be reused with other composition rules.

Listing 3.13 shows how to attach the rule-based *PlugBehavior* with the *OpenSlotRule* on the *Container* slot of the workbench. This rule-based behavior establishes the same composition result as the *DependentSlotBehavior* from Listing 3.10 and Figure 3.25.

```
[Extension]
...
class Workbench : ExtensionBase, IApplication {
  Workbench() {
    Slots["Container"].Behaviors.Add(
        new PlugBehavior(new OpenSlotRule(Slots["View"])));
  }
  ...
}
```

Listing 3.13: Binding a rule-based composition behavior to a slot

In order to allow composition rules to be reused with different behaviors, rule-based behaviors translate their composition events and their composition state into generic composition events and a generic composition state and forward them to their composition rule. The composition rule implements its composition logic based on generic composition events and a generic composition state.

Figure 3.26 on the next page shows how the rule-based *PlugBehavior* cooperates with the *OpenSlotRule* to ensure that the workbench's *View* slot is only opened if a container is plugged: (1) when the *PlugBehavior* receives the *Plugged* event from the *Container* slot, (2) it translates it into a generic *Composed* event and forwards it to the *OpenSlotRule*, which (3) opens the *View* slot. Vice versa, (4) when the *PlugBehavior* receives the *Unplugging* event from the *Container* slot, (5) it translates it into a generic *Decomposing* event and forwards it to the *OpenSlotRule*, which (6) closes the *View* slot. If someone tries to open the *View* slot, (7) the *OpenSlotRule* receives a *CanOpen* event, and (8) checks the generic composition state by retrieving the behavior's *IsComposed* property, which (9) is translated into the *PlugBehavior*-specific *IsPlugged* operation that checks whether a contributor is plugged into the *Container* slot.

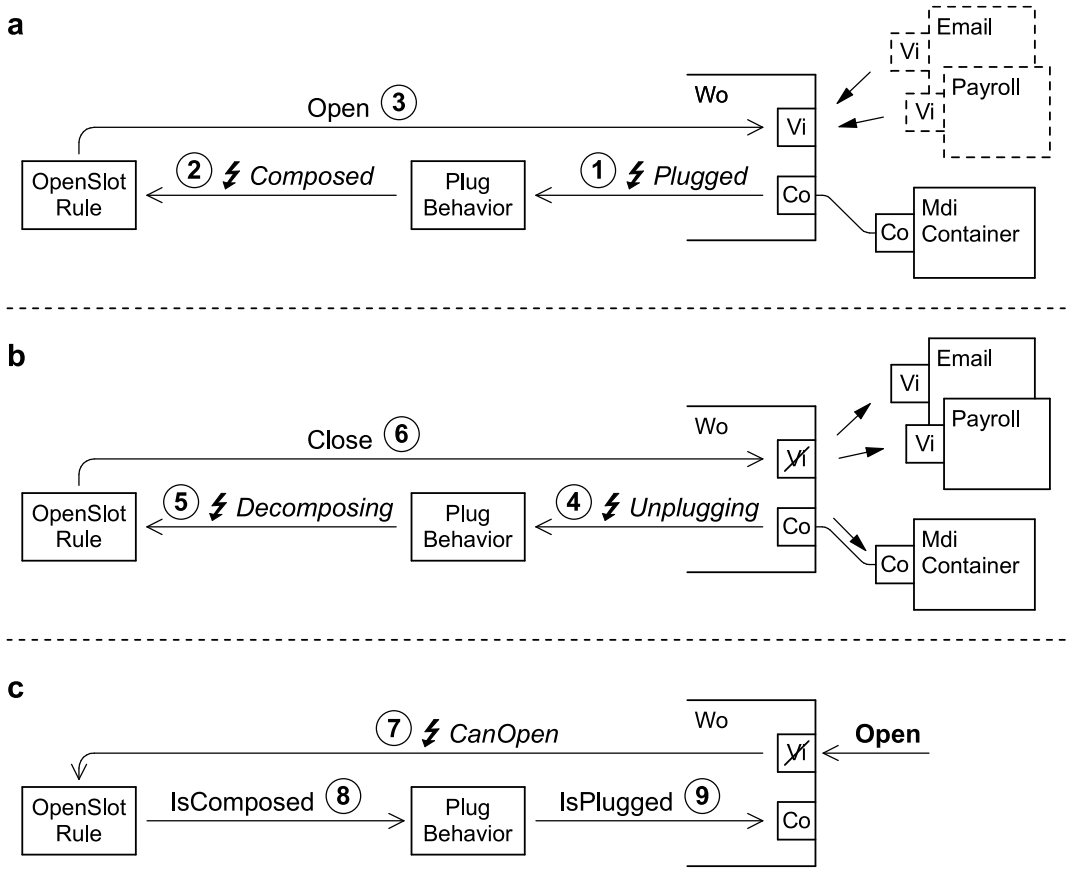
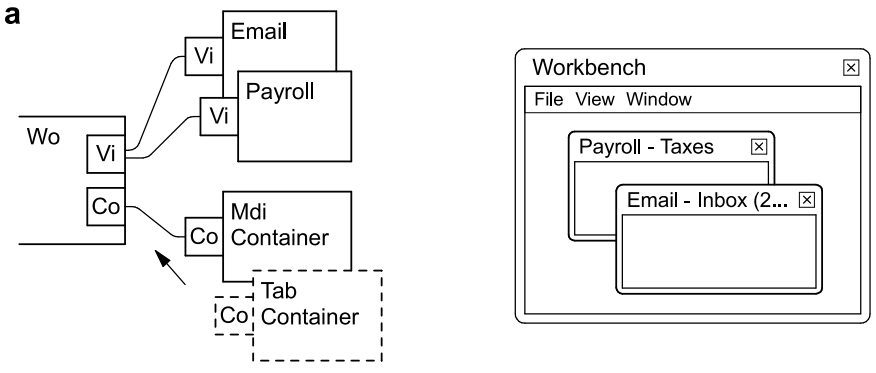


Figure 3.26: Rule-based composition behavior translating composition events and the composition state of a composition operation to a composition rule

Figure 3.27 shows an example of how the *PlugBehavior* can be reused with a different composition rule. As the workbench can only deal with a single container at a time, we want to make sure that the number of plugged containers is limited to one. A solution for this is to unplug an already plugged container, if another container is plugged into the workbench. This can be done with a *PlugBehavior* in combination with a *ReplaceRule* for the *Container* slot: (a) when the *TabContainer* is plugged, the behavior notifies its *ReplaceRule* that a new contributor was composed. In (b) the *ReplaceRule* reacts to the notification and sends a generic *Decompose* command to the *PlugBehavior*, which is translated to an *Unplug* command that finally unplugs the *MdiContainer*.



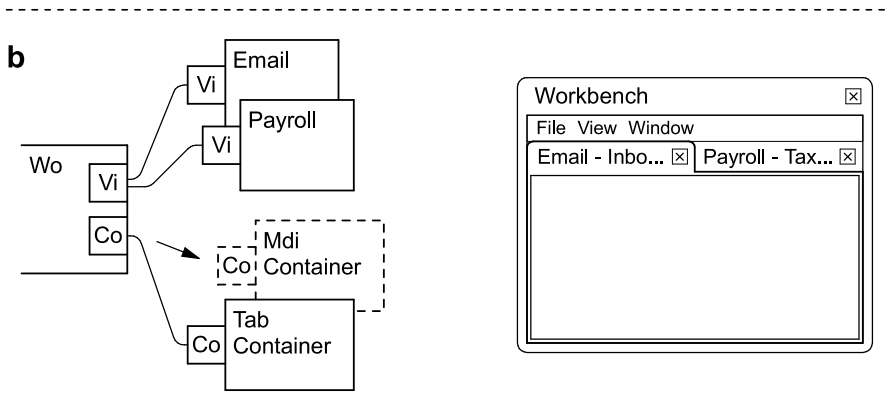


Figure 3.27: A PlugBehavior with a ReplaceRule ensures that there is only one contributor plugged at the same time

Listing 3.14 shows that the workbench can also attach multiple behaviors to its *Container* slot, e.g., a *PlugBehavior* with an *OpenSlotRule* and a *PlugBehavior* with a *ReplaceRule*. The *ReplaceRule* unplugs previously plugged extensions, as soon as new extensions get plugged and the maximum number of plugged extensions is reached. The maximum can be set through a parameter in the rule's constructor. An additional parameter for the replace mode determines if the *ReplaceRule* should unplug previously plugged extensions just before or just after a new extension was plugged. The *ReplaceMode.AfterComposition* specifies that this rule should not perform the replacement until the new extension was plugged. If multiple behaviors are attached to a slot, they are executed in the order in which they were attached. In the example of Listing 3.14, however, the behaviors are independent, thus the order does not matter.

```
[Extension]
...
class Workbench : ExtensionBase, IApplication {
  Workbench() {
    Slots["Container"].Behaviors.Add(
      new PlugBehavior(new OpenSlotRule(Slots["View"])));
    Slots["Container"].Behaviors.Add(
      new PlugBehavior(
        new ReplaceRule<Plug>(1, ReplacementMode.AfterComposition)));
  }
  ...
}
```

Listing 3.14: Attaching multiple composition behaviors to a slot

The example in Figure 3.28 on the next page shows how we can reuse the *ReplaceRule*, however this time in combination with a *TagBehavior*. For this, we disclose a further improvement of our workbench example. In order to use the composition state for keeping track of the current focus view, we define an additional tag for the workbench's *View* slot, namely the *Focus* tag. The view, which is tagged with this tag, has the focus. Since only one view can have the focus at the same time, every time

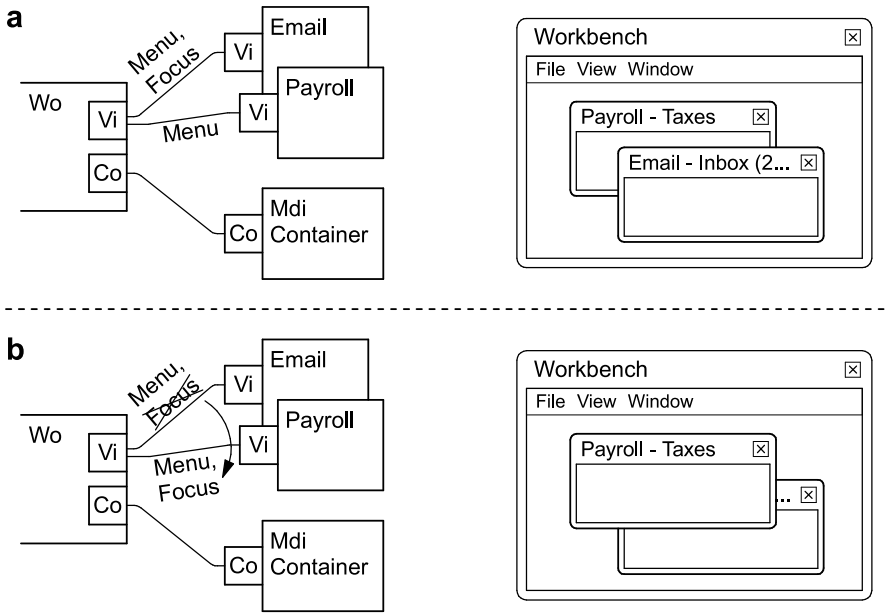


Figure 3.28: Setting the focus of a view by the use of the composition state with the Focus tag

when a new view gets the *Focus* tag, the *TagBehavior* with the *ReplaceRule* removes the previous *Focus* tag. In Figure 3.28 (a) the *Email* view is tagged with *Menu* and *Focus*. As described in Section 3.3.1 on page 45, the *Menu* tag indicates which views should be visible in the view menu, the newly added *Focus* tag specifies that the *Email* view is currently focused. As soon as the *Payroll* view is tagged with *Focus* in (b), the *Payroll* view is focused in the workbench window and the *TagBehavior* with the *ReplaceRule* removes the *Focus* tag from the *Email* view.

The *TagBehavior* with the *ReplaceRule*, should only replace the *Focus* tag but not the *Menu* tag, if multiple views are tagged with *Menu*. Therefore, beside the composition rule, the rule-based *TagBehavior* has a further parameter, which filters the tag to which the behavior should be applied. In Listing 3.15 we attach the *TagBehavior* with the *ReplaceRule* to the *Container* slot and use the "Focus" parameter to set a filter for the *Focus* tag.

```
[Extension]
...
class Workbench : ExtensionBase, IApplication {
  Workbench() {
    ...
    Slots["View"].Behaviors.Add(
      new TagBehavior("Focus",
        new ReplaceRule<Tag>(1, ReplacementMode.AfterComposition)));
  }
  ...
}
```

Listing 3.15: Binding a TagBehavior with a filter for the tag to which the behavior is applied

Listing 3.16 shows the implementation of the rule-based *PlugBehavior*. All rule-based composition behaviors inherit from their base class *RuleBasedBeahvior*, which has a type parameter denoting the meta-object to which the composition operation should apply. The constructor takes a *CompositionRule* and passes it to its base class, where it is stored. Using the *ComposedObjects* property, the composition rule can retrieve the behavior's composition state in a generic way. In the case of *PlugBehavior*, this is the collection of plugs, which are plugged into the behavior slot. If the *CompositionRule* needs to perform a composition operation, it calls the behavior's generic *Compose* or *Decompose* methods, which are translated into the *PlugBehavior*-specific *Plug* or *Unplug* operations. The composition events *CanPlug*, *Plugging*, *Plugged*, *CanUnplug*, *Unplugging*, and *Unplugged* are forwarded to their generic counterparts in the composition rule. The implementation, shown in Listing 3.16, is a simplified version; the real implementation has additional methods to retrieve candidates for composition, i.e., meta-objects that are currently available and could be composed. Composition candidates can be used, for example, in UI-bound composition behaviors, where the

```
class PlugBehavior : RuleBasedBehavior<Plug> {
  public PlugBehavior(CompositionRule<Plug> rule) : base(rule) { }

  override Collection<Plug> ComposedObjects {
    get { return BehaviorSlot.PluggedPlugs; }
  }

  override void Compose(Plug plug) {
    BehaviorSlot.Plug(plug);
  }
  override void Decompose(Plug plug) {
    BehaviorSlot.Unplug(plug);
  }

  override bool CanPlug(CompositionEventArgs args) {
    return Rule.CanCompose(args.Plug);
  }
  override void Plugging(CompositionEventArgs args) {
    Rule.Composing(args.Plug);
  }
  override void Plugged(CompositionEventArgs args) {
    Rule.Composed(args.Plug);
  }

  override bool CanUnplug(CompositionEventArgs args) {
    return Rule.CanDecompose(args.Plug);
  }
  override void Unplugging(CompositionEventArgs args) {
    Rule.Decomposing(args.Plug);
  }
  override void Unplugged(CompositionEventArgs args) {
    Rule.Decomposed(args.Plug);
  }
}
```

Listing 3.16: Implementation of a rule-based composition behavior

composition is controlled by the user via the user interface (see UI-boundend Composition Behaviors below).

The implementation of the *ReplaceRule* is shown in Listing 3.17. Composition rules are derived from the common base class *CompositionRule*. The base class *CompositionRule* maintains a list of composed objects (e.g., contributors in a slot). The class *ReplaceRule* decomposes the composed object that was composed first if the maximum number of composed objects (denoted by the field *maximum*) is exceeded. It has also a field for the replacement mode, which specifies if the rule should fire before, or after a composition event. For this the rule overrides both, the *Composing* and the *Composed* event handlers. Depending on the replacement mode, one of these methods checks whether the maximum amount of composed objects is already reached and calls the method *Decompose* from the base class before or after a composition event if required. The *Decompose* method gets the first composed object as an argument. The base class forwards the call to the composition behavior, which finally translates the generic composition operation into a concrete composition operation, e.g., *Unplug* if the rule is combined with a *PlugBehavior*.

```
enum ReplacementMode {
  BeforeComposition,
  AfterComposition
}

class ReplaceRule<T> : CompositionRule<T> {
  int maximum;
  ReplacementMode replacementMode;

  ReplaceRule(int maximum, ReplacementMode replacementMode) {
    this. maximum = maximum;
    this.replacementMode = replacementMode;
  }

  override void Composing(T metaObject) {
    if (replacementMode == ReplacementMode.BeforeComposition
        && ComposedObjects.Count == maximum) {
      Decompose(ComposedObjects[0]);
    }
  }
  override void Composed(T metaObject) {
    if (replacementMode == ReplacementMode.AfterComposition
        && ComposedObjects.Count > maximum) {
      Decompose(ComposedObjects[0]);
    }
  }
}
```

Listing 3.17: Implementation of a composition rule

The Plux composition library implements a rule-based composition behavior for every composition operation. Additionally, it implements a set of predefined composition rules, which cover common composition patterns, such as limiting the cardinality of

contributors either through replacing or through denying them, ensuring a certain composition order, or filtering contributors depending on certain criteria, e.g., on a value provided by a parameter of a contributor's plug or by the vendor of a contributor.

UI-bound Composition Behaviors

In self-contained composition behaviors, the composition logic is implemented within the behavior itself. In rule-based composition behaviors, the composition logic is extracted into a composition rule. Finally, in UI-bound composition behaviors, the composition logic is provided by the user of an application, i.e., the user triggers composition operations through the application's user interface. Vice versa, the user interface is updated by the behavior on changes in the composition state. Therefore, instead of a composition rule, the constructor of an UI-bound behavior gets an UI control as an argument, which is bound to the composition state by the behavior.

In the example of Figure 3.28 on page 73 the workbench receives the *Tagged* event for the *Focus* tag to focus the tagged view. To ensure that only one view is tagged with *Focus* at the same time, we attached a *TagBehavior* in combination with a *ReplaceRule* to the *View* slot (see Listing 3.15). However, the *ReplaceRule* only ensures that just a single contributor has the *Focus* tag at the same time, but it does not focus the tagged view in the workbench, nor it updates the *Focus* tag in the composition state when the user changes the focus of a view by clicking on a view window in the workbench. This implementation has to be done programmatically in the workbench. In order to extract this implementation from the workbench too, we can use an UI-bound behavior that binds the *Focus* tag to the user interface of the workbench.

Figure 3.29 on the next page shows how a UI-bound behavior works that binds the composition state of the workbench's *View* slot to the workbench's container control. The UI-bound behavior, which is called *ViewBehavior*, is attached to the *View* slot and handles the slot's *Tagged*, *Plugged* and *Unplugging* events. When a view is tagged with *Focus*, the *ViewBehavior* focuses the tagged view in the container, when a view is plugged, the behavior opens it in the container, and when a view is about to be unplugged, it closes it in the container. For this, the *ViewBehavior* also reacts to the UI events *FocusChanged* and *ViewClosed* from the container. The behavior handles those events and updates the composition state accordingly.

In Figure 3.29 (a) the *Email* view and the *Payroll* view are plugged into the *View* slot, therefore both views are opened. As the *Payroll* view currently has the focus, it is tagged with the *Focus* tag. When the user clicks on the *Email* view in (a), in (b) the container focuses the view and the UI-bound behavior moves the *Focus* tag from the *Payroll* view to the *Email* view. When the user closes a view by clicking on its close button in (b), the behavior reacts on the container's *ViewClosed* event and (c) unplugs the view from the *View* slot. Furthermore, as the *Email* view is closed now, the container focuses the *Payroll* view and the behavior sets the *Focus* tag to the *Payroll* view. As the *Email* view is not plugged to any slot anymore, it was deactivated by the

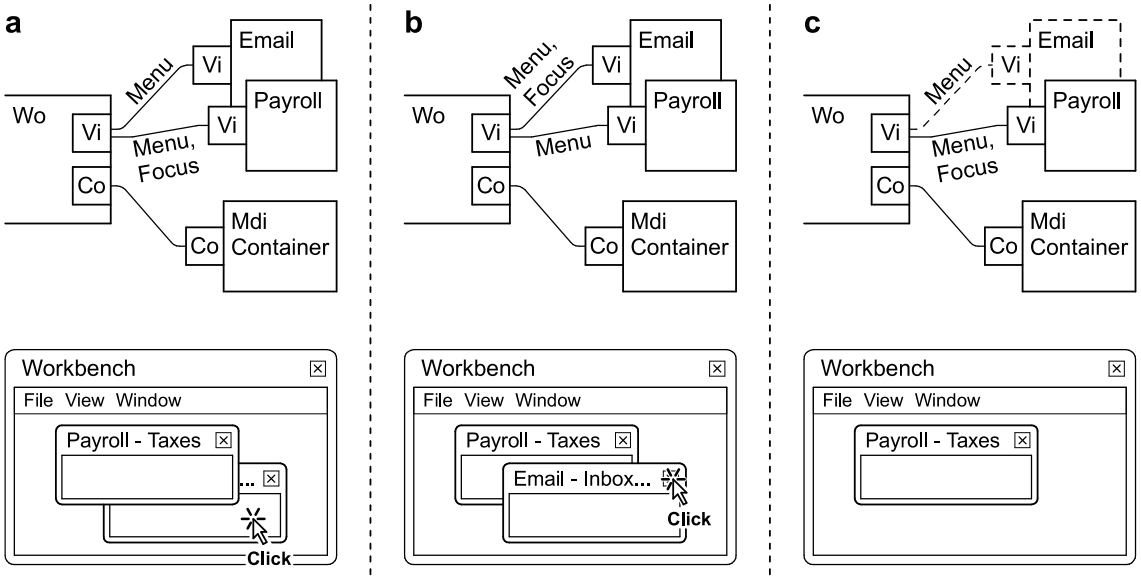


Figure 3.29: Binding the composition state to the user interface of an application

composer. Listing 3.18 on page 80 shows the final implementation of the workbench example, which includes the code that shows how the *ViewBehavior* is attached to the workbench's *View* slot.

As we have bound the composition state of the *View* slot to the workbench's container control, the container can now be controlled via the composition state of the *View* slot. Thus, if we now bind the *View* menu and the *Window* menu to the *View* slot too, views can be controlled via these menus, but without having any dependencies between the menus and the container. Similar to that, of course, views also can be controlled with any other user controls or tools that are bound to the *View* slot.

In Figure 3.30 we use two further UI-bound behaviors to bind the composition state of the *View* slot to the view menu and to the window menu. The view menu displays an entry for every view contributor that is tagged with the *Menu* tag. The user can open or close a view by clicking on these menu entries, i.e., clicking on such an entry either plugs or unplugs a view, while the *ViewBehavior*, which is described above, opens or closes the view in the container. In the view menu, open views are marked with a checkmark. The window menu displays an entry for each open view, i.e., for each plugged view. The focused view is marked with a checkmark, i.e., the window menu marks the view that is tagged with the *Focus* tag. By clicking on an entry in the window menu the user can set the focus to the corresponding view. For this, the UI-bound behaviors for the view menu and for the window menu get a reference to the according menu control via their constructor and add or remove menu items depending on the composition state. Furthermore, these UI-bound behaviors react on the menu items' *Click* event and update the composition state accordingly.

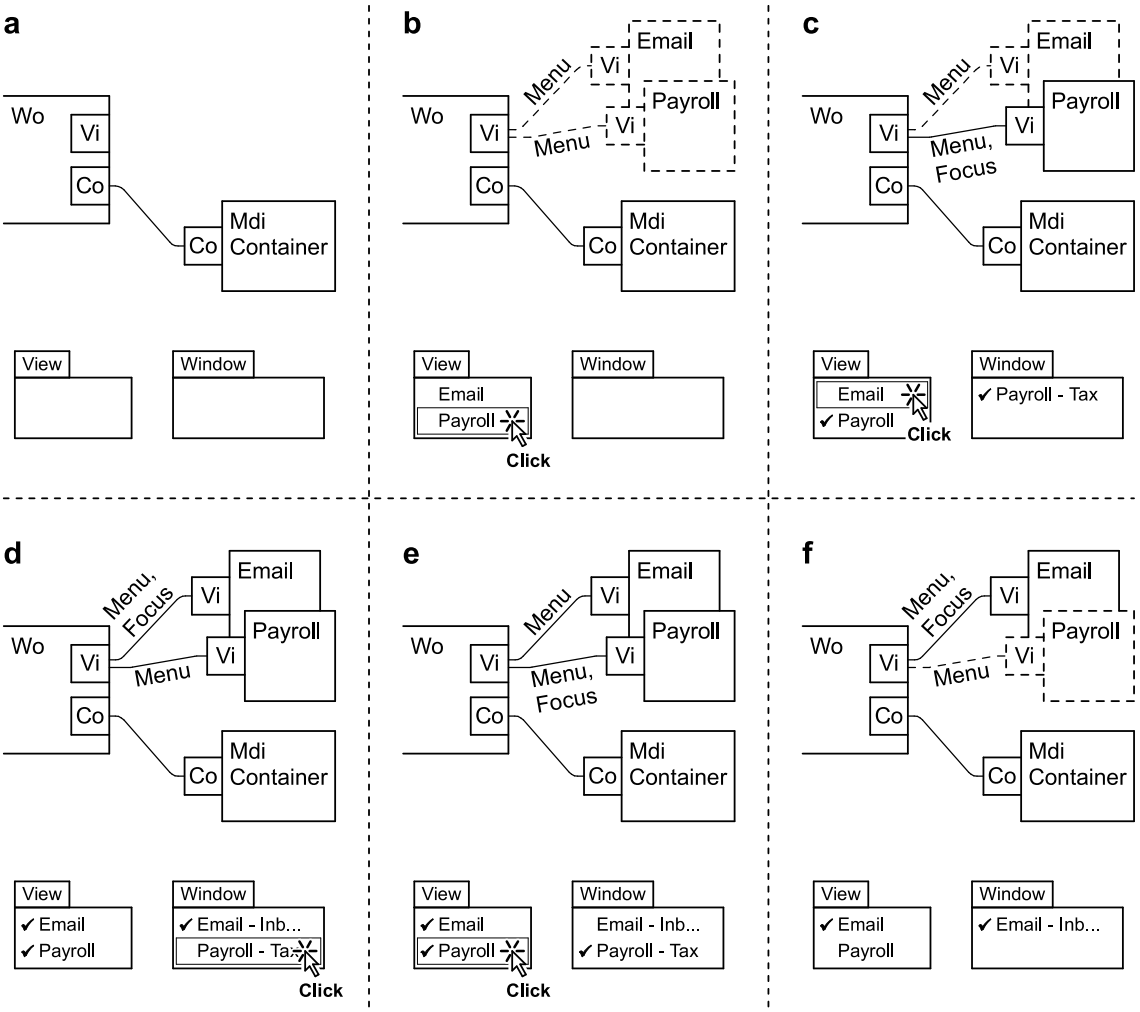


Figure 3.30: Modifying the composition state via the user interface

As in Figure 3.30 (a) no views are tagged or plugged to the *View* slot, the view menu and the window menu are empty. In (b) the *Email* view and the *Payroll* view are tagged with *Menu* and therefore displayed within the view menu. When the user clicks on the view menu's *Payroll* entry, in (c) the behavior plugs the view into the *View* slot and the *Payroll* menu entry is marked with a checkmark. Additionally, the window menu displays an entry for the plugged view. As the *Payroll* view was just opened, it got the focus in the container and therefore was tagged with the *Focus* tag. Thus, the menu entry of the window menu is marked with a checkmark too. When the user clicks on the menu entry for the *Email* view, in (d) the same as with the *Payroll* view happens with the *Email* view. When the user clicks on the *Payroll* entry in the window menu, in (e) the behavior moves the *Focus* tag to the *Payroll* view and thus the container focuses this view. If the user clicks on the menu entry of the opened Payroll view in the view menu, the behavior unplugs the view, which causes the view to be unplugged in (f) and thus closed. As the *Payroll* view is unplugged now, the checkmark for this view in the view menu is removed and the entry in the window menu is removed, too.

Listing 3.18 shows the final implementation of the workbench example, which combines automatic composition, programmatic composition, and behavior-guided composition in order to show plugged views within an exchangeable container and to control views via the view menu and the window menu. The parts, which were not described until now, are highlighted. In the constructor we create the window for the workbench, the view menu, and the window menu. Additionally, we attach a *PlugBehavior* with the *OpenSlotRule* and a *PlugBehavior* with the *ReplaceRule* to the *Container* slot. The *ViewMenuBehavior* binds the composition state of the *View* slot to the view menu and the *WindowMenuBehavior* binds the composition state of the *View* slot to the window menu. In the attribute for the *Container* slot we register the event handler method *ContainerPlugged* and *ContainerUnplugging*. When a container is plugged, the

```
[Extension]
[Slot("Container", Plugged="ContainerPlugged,
                   Unplugging="ContainerUnplugging")]
[Slot("View", AutoOpen=false)]

class Workbench : ExtensionBase, IApplication {
  Window window;
  Menu viewMenu
  Menu windowMenu;
  ViewBehavior viewBehavior;

  Workbench() {
    window = ...
    window.Closed += WindowClosed;
    viewMenu = ...
    windowMenu = ...
    Slots["Container"].Behaviors.Add(
      new PlugBehavior(new OpenSlotRule(Slots["View"])));
    Slots["Container"].Behaviors.Add(
      new PlugBehavior(
        new ReplaceRule<Plug>(1, ReplacementMode.AfterComposition)));
    Slots["View"].Behaviors.Add(new ViewMenuBehavior(viewMenu));
    Slots["View"].Behaviors.Add(new WindowMenuBehavior(windowMenu));
  }

  void ContainerPlugged(CompositionEventArgs args) {
    IContainer container = (IContainer) args.Plug.Extension.Object;
    AddConainer(container.Control);
    viewBehavior = new ViewBehavior(container);
    Slots["View"].Behaviors.Add(viewBehavior);
  }

  void ContainerUnplugging(CompositionEventArgs args) {
    IContainer container = (IContainer) args.Plug.Extension.Object;
    Slots["View"].Behaviors.Remove(viewBehavior);
    viewBehavior = null;
    RemoveContainer(container.Control);
  }

  void AddContainer(Control containerControl) { /* not shown*/ }

  void RemoveContainer(Control containerControl) { /* not shown*/ }
```

```
  void Start() {
    window.Show();
  }
  void WindowClosed() {
    Extension.Destroy();
  }
}
```

Listing 3.18: Final implementation of the Workbench extension using automatic, programmatic, and behavior-guided composition

event handler *ContainerPlugged* adds the container control to the window control and creates a *ViewBehavior* that binds the composition state of the *View* slot to the currently plugged container. Vice versa, when a container is about to be unplugged, the event handler *ContainerUnplugging* detaches the *ViewBehavior* from the *View* slot and removes the container from the window. When the *Workbench* is plugged into the Plux core's *Application* slot, the method *Start* is called. *Start* opens the window for the workbench. When the window is closed, the event handler *WindowClosed* uses programmatic composition to destroy the *Workbench* extension, which includes decomposing all its contributors.

For simplicity reasons, we omitted some details of the real workbench implementation. For example, we did not show that the menu is implemented as an extension itself, which can be extended with pluggable menu entries. Furthermore, we did not cover how we handle the customizable order of menu entries, which is implemented with an order parameter that must be provided by the plug of each view and the plug of each pluggable menu entry.

3.3.7 User-guided composition

Developers can influence the composition process using programmatic composition and composition behaviors. However, with user-guided composition even administrators and users can control the composition of an application. Since Plux maintains the composition state, composition tools can retrieve the composition state, present it to the user, and modify it. Plux ships with several composition tools, such as a *Visualizer*, which presents the composition state in a graphical manner; a *Console*, which provides text-based access to the composition state, a *Persistor*, which persists and restores the actual composition state; or a *Scripting Engine*, which allows the execution of predefined composition scripts. Furthermore, we implemented some experimental composition tools for user-guided composition that compose an application based on the available hardware, e.g., we implemented a game, which gets recomposed based on the pieces on a playing field.

Visualizer

The *Visualizer* is a workbench view, which draws a graph of the current composition state with extensions, slots, plugs, and their connections. It supports developers, administrators, and users in understanding the composition of an application. Furthermore, the visualizer allows the user to click on each meta-object representation in the graph in order to see detailed information about the meta-object's state and to modify its composition state.

Figure 3.31 shows how the visualizer presents a composition state of an application (note, that this application happens to contain the visualizer itself as an extension). In the example, the user clicked on the *View* slot of the *Workbench* extension to see its options. In *Properties* the user can retrieve and modify the composition configuration of the slot, for example, he can enable or disable automatic composition for this slot. With the composition operations below, the user can modify the composition state. As the *View* slot is open, the *Open* composition operation is disabled. For the operations *Tag*, *Untag*, *Plug*, and *Unplug*, the visualizer provides a list of all possible contributors, to which the operation can be applied. As the *Visualizer* view is the only plugged contributor in the *View* slot, the *Unplug* operation lists this contributor as the only possible candidate to be unplugged.

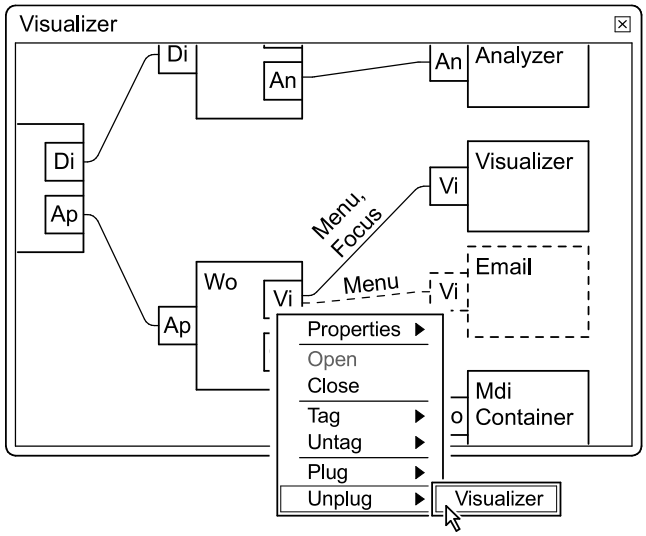


Figure 3.31: The visualizer presents the current composition state in a graphical manner and allows users to modify the composition state

Console

The *Console* provides a text-based interface to the composition state of an application. It supports commands to retrieve and to modify the composition state. In order to make the console customizable and extensible, its commands are implemented as separate extensions, which are plugged to the console via its *Commands* slot.

In Figure 3.32, the console view is extended with two *Commands* contributors. While the *RuntimeCommands* implements commands for retrieving and modifying the composition state, the *LayoutCommands* extend the Console with commands for modifying the arrangement of controls in views that use the Plux *Layout Library* (see Section 6.3 Runtime Libraries) to build a component-based user interface from control extensions. The example in Figure 3.32 shows the *get-extension* command that lists all extensions, which match an optional filter, and the *plug* command that performs the *Plug* operation for a host and a contributor.

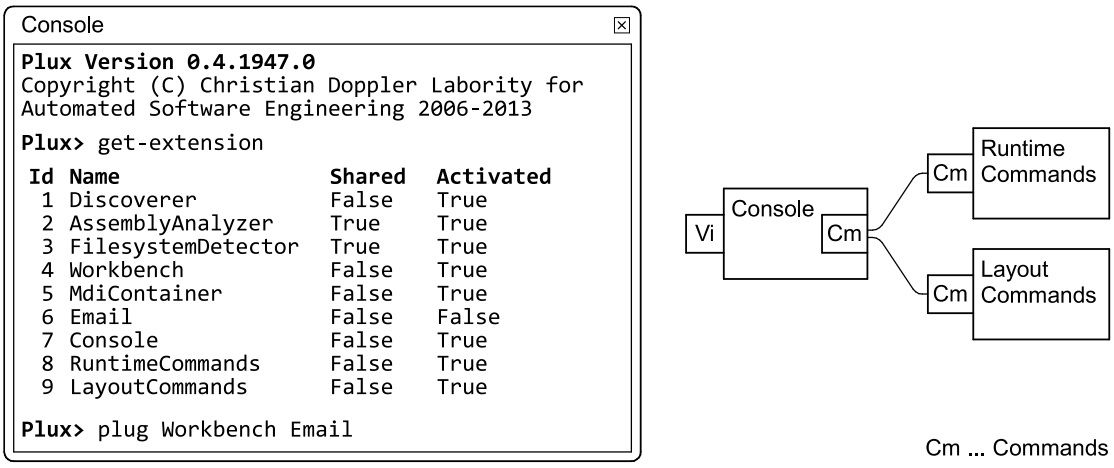


Figure 3.32: The console provides a text-based interface to the composition state of an application

Persistor

The *Persistor* saves the current composition state, in order to restore it at a later time. This is useful to save the composition state at one time and continue work at another time with the same composition as before. The persistor can also be used if an application error occurs. It can then save the actual composition state to a file and transfer it to the vendor of the application, so that the vendor can reproduce the user's composition state on his own machine and look at the error.

Scripting Engine

The *Scripting Engine* allows executing scripts that perform composition operations in order to establish a certain composition state of an application. Applications can thus be easily reconfigured at run time to customize them for a current working task. Scripts can be triggered, for example, by clicks on the user interface or by starting them via the Plux console.

3.4 Interaction Standard

The interaction standard specifies how extensions communicate among each other, and how they exchange data. As extensions can retrieve references to the extension object of other extensions, hosts can call their contributors' methods directly. However, developers must not expect that the referenced extension object is an instance of the class that was marked with the extension attribute. In some situations, e.g. when Plux is started with certain security constraints, Plux uses proxy objects instead of the original extension objects. Proxy extension objects forward calls to the actual implementation of an extension.

3.4.1 Thread Management

The Plux interaction standard specifies a dedicated runtime thread, in which all operations concerning the component model must be performed. Thus, extensions must be executed in the runtime thread if they retrieve the composition state, if they call methods of other extensions, and if they perform composition operations in programmatic composition. As Plux raises all composition events in the runtime thread, and as method calls beyond extension boundaries must be performed within the runtime thread, developers need not worry about thread synchronization issues, such as race conditions or deadlocks and can be sure that the composition state is not modified unexpectedly from a background thread.

If an extension implements a long running task using a worker thread, it must ensure that the worker thread does not escape the extension's boundaries. Such extensions can use the Plux dispatcher to dispatch their results back into the runtime thread. Furthermore, extensions that listen to events that are raised from other threads than the runtime thread (e.g., file system events from the operating system) can use the dispatcher to handle such events in the runtime thread. The dispatcher provides an interface for checking whether the current code is executing in the runtime thread and to dispatch calls either synchronously or asynchronously into the runtime thread.

3.4.2 Exception Handling

Plux ensures that its runtime does not crash if any unhandled exception occurs. As Plux performs all operations in the runtime thread, each executed code was initiated from a dispatcher operation. In the case of an unhandled exception, the dispatcher operation gets aborted, the runtime logs the exception, and raises an *UnhandledException* event. Afterwards, the dispatcher continues its work with the next dispatcher operation in the queue.

3.5 Customization Standard

Extensions consist of code and metadata, which are discovered by the discoverer, and which cannot be modified during run time. However, the Plux customization standard defines a settings model for extensions, by which extensions can be enriched with configurable settings. Extension settings are discovered by an exchangeable settings discoverer and can be retrieved via the meta-object of an extension. Vice versa, extension settings can be modified, whereby modifications are written back to the settings source. If an extension writes to its settings, but no settings were discovered, the setting discoverer creates a settings source on demand. The Plux infrastructure implements a settings discoverer for XML files and a settings discoverer that retrieves the settings from a database.

Listing 3.19 shows an example for an XML settings file. The example shows the settings for the *Visualizer* extension, which is described in Section 3.3.7 on page 80. The example file only contains settings for a single extension; however, every setting file can contain settings for multiple extensions from different plugins. The *plugin* element's *name* attribute references the plugin *Plux.Visualizer* and the *name* attribute of the child element *extension* references the *Visualizer* extension, which is implemented in the *Plux.Visualizer* plugin. Settings are specified with the *setting* element, which maps from a key to a value, or with the *list* element, which maps from a key to a list of values, or with the *dictionary* element, which maps from a key to a dictionary of settings. Setting dictionaries again can contain *setting* elements, *list* elements, and further *dictionary* elements. Values can have any simple type such as *Boolean*, *Integer*, or *Double*, as well as types for which a type converter is implemented.

```
<?xml version="1.0" encoding="utf-8"?>
<settings xmlns="http://ase.jku.at/plux/SettingsSchema/">
  <plugin name="Plux.Visualizer">

    <extension name="Visualizer">
      <setting key="ShowDisplayNames" value="True" />
      <setting key="ShowExtensionId" value="False" />
      ...

      <list key="IgnoreExtensions" >
        <item value="Console" />
      </list>

      <dictionary key="DisplayNames">
        <setting key="Application" value="Ap" />
        <setting key="Workbench" value="Wo" />
        ...
      </dictionary>
    </extension>

  </plugin>
</settings>
```

Listing 3.19: XML settings file for the *Visualizer* extension

Listing 3.20 shows how the *Visualizer* extension uses the settings, which are specified in Listing 3.19, to configure the appearance of its graph. Extension settings are retrieved via the *Settings* property of the extension's base class *ExtensionBase*. The property *ShowDisplayNames* specifies, if the graph of the Visualizer should display the original extension names, or if a special display name should be used instead, e.g., using the display name "*Wo*" instead of the extension name "*Workbench*". As the value of this property is specified in an extension setting, it is retrieved via the *Settings* property. The method *GetValue* gets a key and a default value as arguments. The key maps to the stored value. The default value is returned, if the key was not found. Furthermore, the default value provides the return type for the generic *GetValue* method. The setter of *ShowDisplayNames* uses the *SetValue* method to store the assigned value back to the settings source.

The *DisplayNames* property returns a dictionary, which stores the mappings from original extension names to their display names. The dictionary is retrieved using the *GetDictionary* method. If the key "*DisplayNames*" was not found, a new dictionary is

```
[Extension]
...
class Visualizer : ExtensionBase, IView {
  ...
  bool ShowDisplayNames {
    get { Settings.GetValue("ShowDisplayNames", true); }
    set { Settings.SetValue("ShowDisplayNames", value); }
  }
  SettingDictionary DisplayNames {
    get { Settings.GetDictionary("DisplayNames");
  }
  SettingList IgnoreList {
    get { Settings.GetList("IgnoreList"); }
  }
  void SetDisplayName(Extension e, String displayName) {
    DisplayNames.SetValue(e.Name, displayName);
  }
  void IgnoreExtension(Extension e) {
    IgnoreList.AddValue(e.Name);
  }
  void Draw() {
    ...
    foreach(Extension e in extensions) {
      if (IgnoreList.Contains(e.Name)) { continue; }
      String extensionName = ShowDisplayNames
        ? DisplayNames.GetValue(e.Name, e.Name) : e.Name;
      ...
    }
  }

}
```

Listing 3.20: Retreiving and modifying extension settings

created. The *IgnoreList* property returns a list of extension names for extensions that should be hidden in the graph.

The method *SetDisplayName* stores a new display name for an extension. As settings dictionaries keep track of modifications, any modifications are written back to the settings source. Similar to this, the method *IgnoreExtension* stores extension names in a *SettingList*. Modifications are written to the settings source, too.

Finally the *Draw* method uses the settings: it uses the *IgnoreList* property to skip all extensions whose names are in the list, it uses the *ShowDisplayNames* property to decide whether to show the original extension name or the display name, and it uses the *DisplayNames* property to retrieve the display names for extensions, whereat the extension name is both, the key and the default value for the *GetValue* method.

Chapter 4

Plugin-based Distributed Multi-user Web Applications

This chapter presents the idea of applying the plugin approach to web applications in order to enable them to be extended and customized by the end users. Users should be enabled to adapt web applications for their specific needs and use their own set of components. User-specific components can be installed either on the server-side in an individual user scope, or on the client-side (i.e., on the user's computer), by which a web application becomes a distributed web application. A case study demonstrates the benefits of building extensible web applications in several usage scenarios.

Web applications face similar problems as desktop applications: if they get big and feature-rich, they become hard to understand and difficult to maintain. Current web applications are hardly customizable and usually not extensible by end users. Furthermore, they cannot access the local hardware of client computers. In order to solve these problems, we applied the plugin approach also to web-based software. While the original version of Plux targeted single-user desktop applications, this thesis presents a number of enhancements so that Plux can now also be used to build *plugin-based distributed multi-user web applications*.

- *Plugin-based.* Plux allows building extensible web applications. Extensions can either be installed by the administrator or even by the end user. Depending on their type of integration, extensions can be classified as: a) *Server-side extensions* that are installed and executed on the server, b) *Client-side* extensions that are installed and executed on the client, and c) *Sandbox* extensions that are installed on the server, but transferred to the client on demand to be executed there in a sandbox.

- *Distributed.* Plux composes extensions into a coherent web application, regardless of if they are executed locally on the web server, remotely on the client-side computer, or remotely on a different server. Whether extensions are installed locally or remotely, they are implemented in the same way and thus

the same extension can be reused in different environments. This gives developers and users a seamless experience.

- *Multi-user.* Plux maintains different user scopes. Extensions can be made available for *all users* of a web application, for a *group of users*, or just for a *single user*. Thus every user can have an individual set of components, i.e., an individual composition state. Authorized users can install their extensions on the server, while non-authorized users can still extend a web application by installing extensions on the client.

- *Web application.* Plux provides an infrastructure for hosting component-based applications on a web server so that they can be accessed via a web browser. The infrastructure cares about thread and session management and provides a lightweight web UI library for building distributed user interfaces, where user controls can be executed on different computers.

We describe several usage scenarios that demonstrate the benefits for extensible web applications. As a running example we use a time recorder web application. The usage scenario covers the distribution of extensions as server-side, client-side, and sandbox extensions, as well as the *multi-user support* via individual user scopes.

The time recorder can be used to record and evaluate working hours. Figure 4.1 shows the basic version of its composition state. Features are implemented as extensions, and the *TimeRecorder* extension is the host for the main features. The basic version consists of two features: one for recording working hours and one for computing and displaying statistics for recorded working hour. For both features, the implementation of the business logic is separated from the user interface. The *RecorderControl* extension provides the user interface that allows the user to start and stop the *Recorder* extension. The *Recorder* is plugged into the *RecorderControl*, generates time records, and stores them using the *DataStore* extension. As the *DataStore* is a shared extension, which is plugged into the *Recorder* and into the *Statistics* extension, the *Statistics* extension can retrieve time records and can compute the statistics that are queried and displayed via the *StaticsControl*. The *StatisticsControl* renders the user interface for the statistics feature and is plugged into the *TimeRecorder* host.

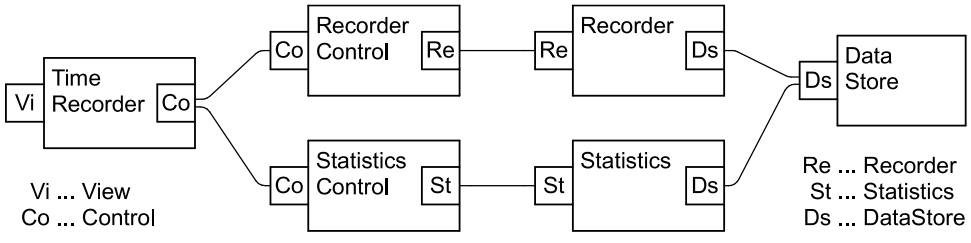


Figure 4.1: Base composition of the time recorder web application

The *TimeRecorder* extension is a view that can be plugged into the *Web Workbench* extension, which is very similar to the workbench described in Chapter 3. However, to keep things simple, extensions that are irrelevant for the usage scenarios (e.g., *Discoverer* or *Workbench*) are not shown in the figures of this chapter.

Figure 4.2 shows the user interface of the time recorder web application. The *RecorderControl* provides buttons for starting, stopping, and pausing records; it also displays the current date as well as the start time of the current record. The *StatisticsControl* shows the time records that match to a selected filter, and also shows the result of a statistical value, which can be selected via the statistics button.

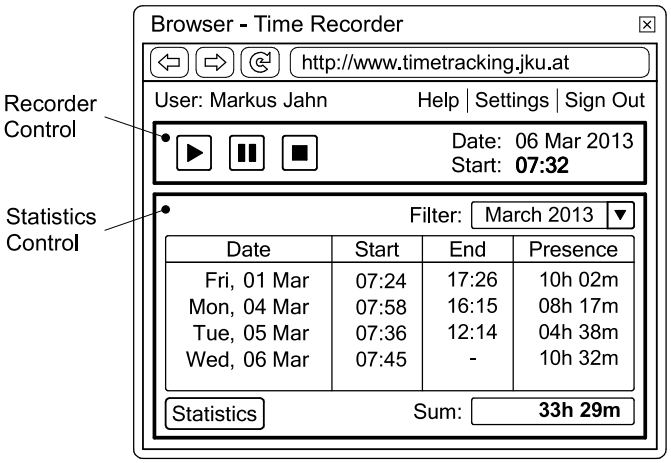


Figure 4.2: User interface of the time recorder web application

As the time recorder is extensible, the user interface must be extensible as well. Control contributors, such as *RecorderControl* and *StatisticsControl*, declare their desired size and position in their metadata. The *Plux Layout Library*, which is implemented by the Plux composition infrastructure, retrieves these layout values and arranges the controls accordingly.

The composition shown in Figure 4.1 is the base configuration of the web application, which is available to all users. All these extensions are server-side extensions, i.e., they are installed and executed on the web server. In the following sections, we show how users can extend this web application with user-specific extensions. We describe how server-side, client-side and sandbox components can be integrated and explain for which scenarios they are suitable.

4.1 Server-side Extensions

Let us assume that a user is not satisfied with the statistics that are provided by the *Statistics* extension. Thus, he can implement a user-specific custom extension, which provides the required functionality. In order to allow the user to access his individual statistics from any computer, the new extension is installed on the server. As the

extension is a user-specific extension, only the user who installed the extension should have access to it and thus it is executed in an individual user scope.

Figure 4.3 shows the user-specific composition for the user *Markus*, who installed the server-side extension *CustomStatistics*. The figure comprises the base composition (shown in Figure 4.1) extended by the new extension. The dashed border indicates the user scope to which the newly added extension is applied.

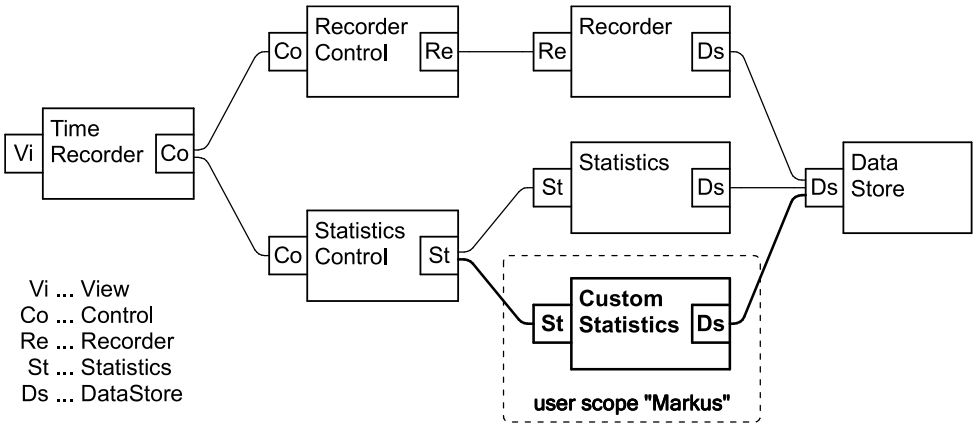


Figure 4.3: Extending a web application with a user-specific server-side extension

Server-side extensions are installed and executed on the server. Thus, they are available all the time, regardless from which computer the user accesses the web application. Because server-side extensions are executed on the same server as all other extensions of the time recorder, there is no performance penalty caused by remote communication. However, extensions in different user scopes are executed in separate memory areas (i.e., in different AppDomains [Microsoft, 2013b]) and thus communication between them causes some performance overhead. As server-side extensions increase the work load on the server and as they may execute malicious code, users typically need to be authorized to install extensions on the server.

4.2 Client-side Extensions

Now we assume that the user *Markus* is an engineer in the field. He needs to track his working hours using a portable device. Because the device cannot connect to the internet, he periodically has to synchronize it with the time recorder application. To synchronize time records, the user connects his device to his office computer, where the client-side extension *MobileSync* has been installed. Because this extension is executed on the client computer, it can access the portable device there.

Figure 4.4 shows the composition for the user *Markus*. To synchronize the data between the device and the time recorder web application, the *Data Store* extension is plugged to the *MobileSync* extension.

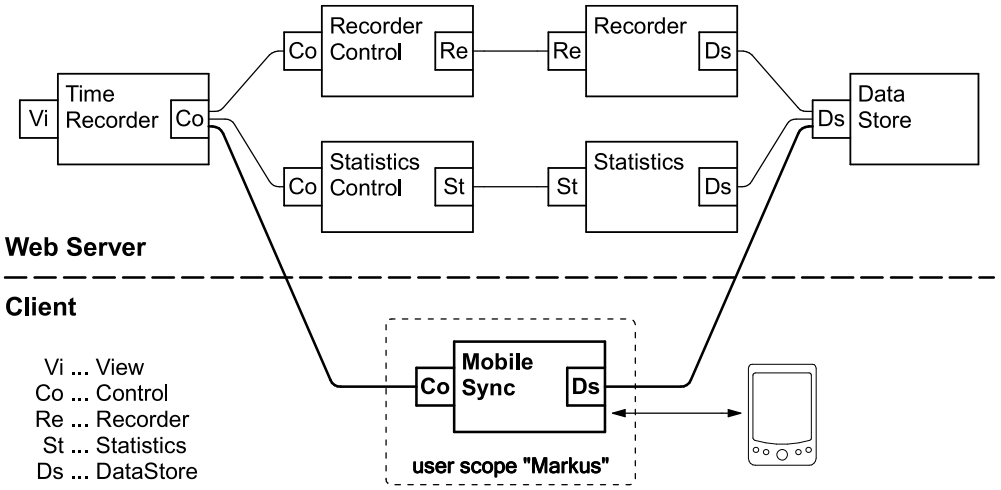


Figure 4.4: Extending a web application with a user-specific client-side extension

Client-side extensions are installed on the client and are remotely plugged into the web application. To plug remotely means that the host and the contributor are executed on different computers. Plux creates proxies on both sides on-the-fly. These proxies handle the communication between the host and the contributor transparently. Note, that Plux allows any extension to run remotely without extra coding effort.

Client-side extensions allow users to build components that integrate local hardware or software into the web application. Furthermore, since client-side extensions are installed on client computers, they enable users to extend their web applications without being authorized to install extensions on the server. However, client-side extensions are executed remotely and cause additional communication overhead.

The above scenario described how a client-side extension enabled a single user to connect his local hardware to a web application. The next scenario now describes a situation, where a group of users need to have access to hardware, which is not located near the web server.

Users, which are in the group *Worker*, do not have the permission to use the web interface for recording their working hours. Instead they must use a hardware time clock to track their working hours. The time clock is connected to a computer on which a client-side extension is installed that integrates the time clock into the time recorder application. Please note, that in this scenario the client-side extension is installed on a different computer than the one that is used to access the web application via the web browser (e.g., to check statistics).

Figure 4.5 on the next page shows the composition for members of the group *Worker*. As these users should only be able to record working hours via the hardware time clock, the server-side extension *RecorderControl* is removed for this group, while the client-side extension *HardwareRecorder* is installed instead. Thus, if a user of this group uses the time recorder from any computer, he will see the user interface of the *HardwareRecorder* (see Figure 4.6), which displays the current status of the *Recorder*, but

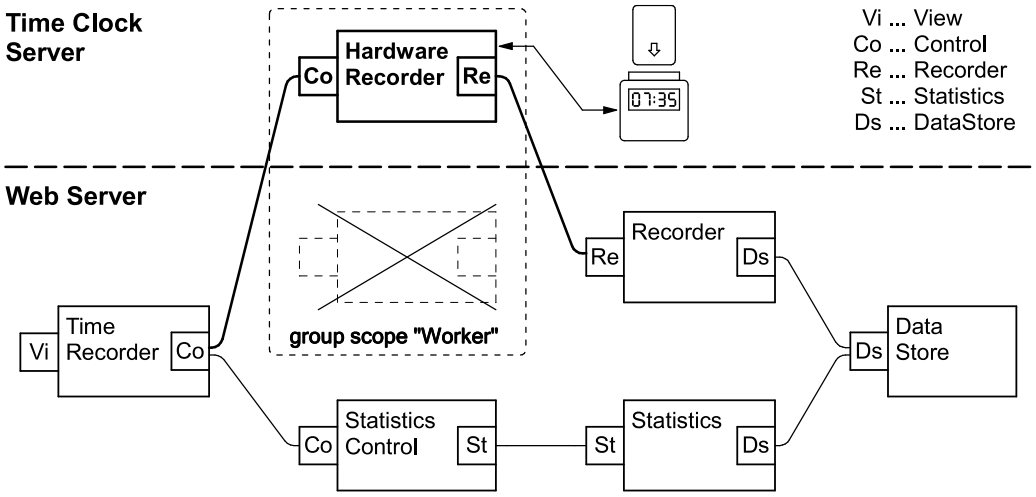


Figure 4.5: Extending a web application with a client-side extension for multiple users

does not allow the user to modify its state. To ensure that the *HardwareRecorder* extension is available for all users in the group at any time, the computer on which the extension is installed is permanently connected to the web server. As this computer acts as a server for the time recorder, the environment in which the hardware time clock is located is called *Time Clock Server*.

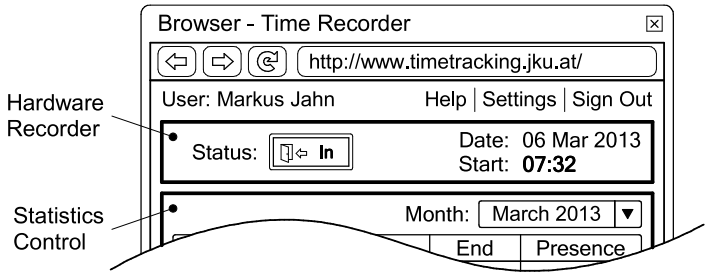


Figure 4.6: User interface of *HardwareRecorder* extension that is executed on a remote computer

Another possible reason for using client-side extensions with group scope instead of a server-side extension is authorization. Even if one is not authorized to install an extension on the server, one can make it available to multiple users as a client-side extension with group scope.

4.3 Sandbox Extensions

In the next scenario, the developers of the time recorder want to provide a richer user interface, e.g., one that is built with *Silverlight* [Microsoft, 2011b] instead of HTML. Silverlight code runs in a sandbox within the web browser of the client. Therefore, the best way to integrate such code into a Plux application is to implement Silverlight

components as extensions that reside on the server but are downloaded to the web browser on demand to be executed there.

In order to implement this scenario, we remove the user interface extensions *RecorderControl* and *StatisticsControl*, implement new UI extensions in Silverlight and install them on the server. There, they are discovered as sandbox extensions. When a user starts the time recorder, the sandbox extensions are downloaded from the server to the client computer and are executed there in the sandboxed Silverlight environment. The business logic extensions remain on the server and are remotely plugged into the Silverlight extensions on the client (see Figure 4.7).

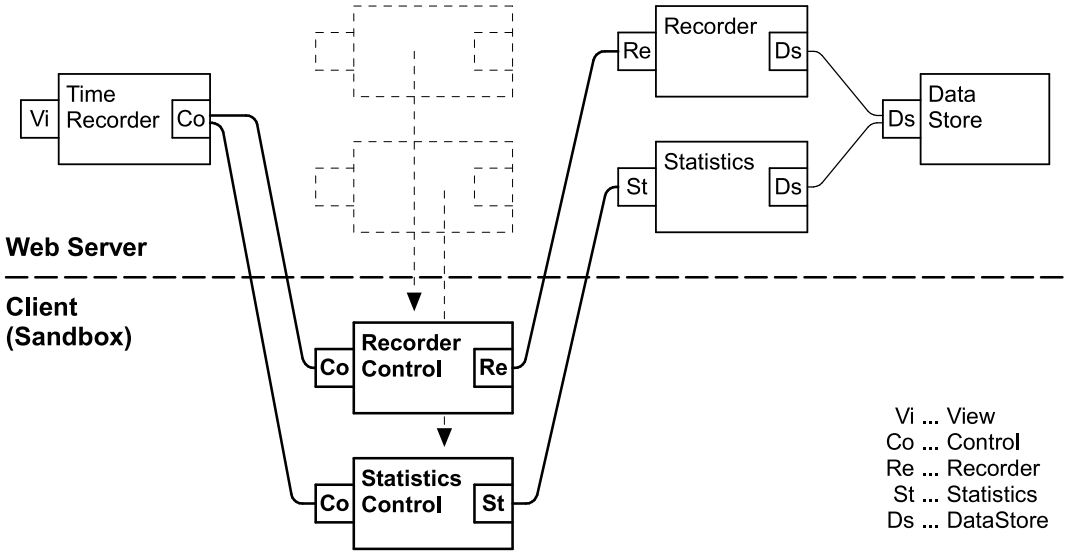


Figure 4.7: Extending a web application with sandbox extensions that are installed on the server, transferred to the client on demand, and executed there in a sandbox

The advantage of sandbox extensions is that they are installed on the server-side, but executed on the client. Thus, they are available for each client, but do not increase the work load on the server. The disadvantage is that the user has to install the Silverlight runtime environment on its computer, which is not installed by default on most operating systems. Furthermore, as such extensions are plugged remotely, they cause additional communication overhead.

4.4 Concluding Example

Finally, Figure 4.8 shows the composition state of a concluding example, which combines all types of extension integration: *server-side*, *client-side*, and *sandbox* extensions. We assume that user *Markus* needs his custom statistics, he tracks working hours with a portable device, he is member of the group *Worker*, so he uses the hardware time clock for recording working hours in the office, and the time recorder application provides a rich user interface that is implemented with sandbox extensions.

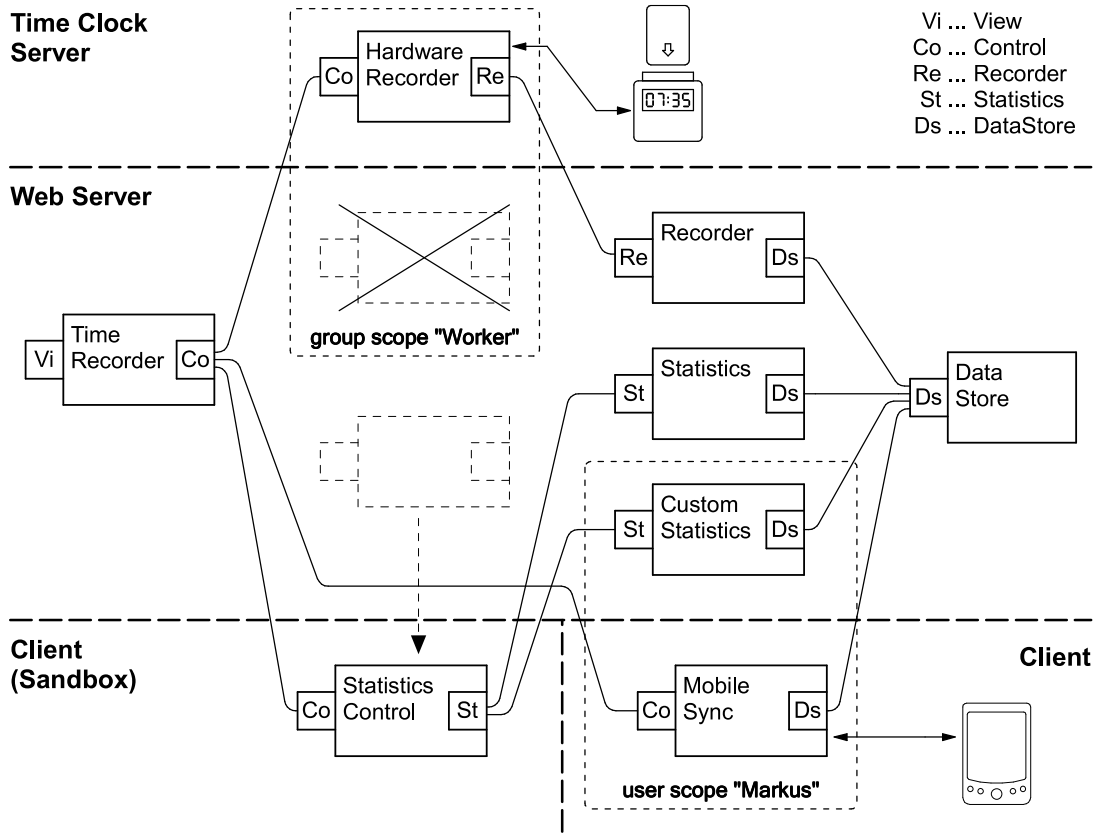


Figure 4.8: User-specific composition composed by server-side, client-side and sandbox extensions

The composition is distributed across three different computers: the *Web Server*, the *Time Clock Server*, and the *Client* computer. Even though extensions are executed remotely on different computers, they are implemented in the same way as extensions that are plugged locally. Developers just declare the extensions' requirements and provisions using metadata (see 3.1 Metadata Standard), and Plux automatically composes them to a coherent web application. Furthermore, Plux handles the communication between the host and the contributor transparently so that developers need not have to care about the distribution of extensions. In other words, a server-side extension of one web application can be reused as a client-side extension in some other web application, and vice versa. There is one exception, though: since the current version of Plux is implemented in .NET, it uses the Silverlight technology for sandbox extensions. Unfortunately, Silverlight assemblies are not binary compatible to .NET assemblies and thus, sandbox extensions need to be compiled in a special way and cannot be reused as server-side or client-side extensions.

This section showed scenarios for the integration of server-side, client-side, and sandbox extensions into a coherent web application. The next section explains the extended component model for the web, which enriches the base component model to support *plugin-based distributed multi-user web applications*.

Chapter 5

The Extended Plux Component Model for the Web

This chapter describes the extended Plux component model for the web. The extended deployment standard specifies user-specific repositories as well as a hierarchical discovery mechanism for local and for distributed plugins. The extended composition standard specifies an individual composition state per user and a distribution mechanism that makes the distribution of plugins transparent to developers and to users. The extended interaction standard specifies the communication between distributed extensions, including distributed thread management, object data synchronization, and lifetime management for distributed objects. As web support, multi-user support, and distribution support is provided by the component model implementation, but not by the developer, the metadata standard and the customization standard remain the same as in the base component model.

The extended Plux component model for the web provides a set of specifications, which enables Plux to build plugin-based distributed multi-user web applications, as described in several usage scenarios in Chapter 4. The specifications of the extended component model add to the specifications of the base component model described in Chapter 3. Thus, everything that is specified in Chapter 3 still is valid in the extended component model for the web.

Plux for the web combines the advantages of the component approach, the distribution approach, and the web approach into a single coherent component model. It aggregates the capabilities of these technologies to provide a more powerful technology than developers would get, if they would combine these technologies independently. The Plux component model for the web distinguishes itself from others by the following characteristics:

Extensible web applications. Existing component technologies, such as CORBA, COM+, Eclipse, OSGi, or MEF, do not target extensible web applications and thus they are not supported. Although the Remote Application Platform (RAP) provides web support for Eclipse, and in spite of the fact that other component technologies can be combined with web application frameworks such as Java EE or ASP.NET to build component-

based web applications, in both cases the composition of a web application cannot be customized individually per user. Plux maintains an individual composition state per user and thus enables users to extend and to reconfigure their web applications for their specific needs.

Distribution and deployment. Many current component technologies and web application frameworks provide support for component distribution. However, Plux stands out in the way it deploys, composes, and interacts with distributed components. In other technologies, deploying components requires developers or administrators to register components in a registry, e.g., the windows registry for COM+, the RMI registry for Java RMI, or a UDDI registry for web services. Moreover, in technologies such as remoting, web services, or OSGi Remote Services, the developer needs to implement particular access points in order to export a service to a distributed computer. In Plux, the implementation of extensions remains the same, regardless of if they are plugged locally on the same computer or remotely on other computers. Deploying extensions in Plux just requires copying the respective plugins into a specific directory, while the discovery mechanism integrates their extensions automatically into the application and makes them available to other local and remote extensions without any extra programming or configuration effort.

Automatic composition. In web application frameworks, the composition has to be done programmatically, and in most component systems the composition is done by configuration. In Plux, the composition is done automatically by the composer, which uses the extensions' self-contained metadata to retrieve requirements and provisions and to connect matching extensions automatically, even if they are distributed over multiple computers. For this, the extended composition standard specifies an automatic distributed composition process.

Automatic lifetime management. Current component technologies, which support distributed components (e.g., CORBA, COM+, or OSGi), as well as technologies for distributed computing (e.g., remoting or web services) do not track which components are connected to each other. Thus, lifetime management usually resorts to one of the following solutions: (1) either the application creates a new instance for an exported service on every request and destroys it immediately afterwards, or (2) the application provides a single instance (or a pool of instances) for an exported service, which is created at startup time and exists as long as the service is available, or (3) the lifetime of an exported service depends on a specified lease time, or (4) the exported service supports distributed reference counting. The first three solutions provide automatic lifetime management, i.e., the consumer of a service does not have to do additional programming for lifetime management. However, if a service is created and destroyed on every single request, it cannot preserve a state during sequential requests, i.e., such a service is stateless. If a service is implemented by using just a single instance, it is shared among all consumers and thus does not have a user-specific state. However, if a service needs to be stateful, this requires an individual instance for every consumer. Thus, the lifetime management of a service either depends on a lease time, which may

expire too soon, or the programmer needs to take care of the lifetime management using reference counting. In Plux, the composition state maintains the instances of extensions and their connections and thus reflects which extensions are in use and which are not. This allows Plux to provide a distributed garbage collection mechanism that destroys unused distributed extensions without the need for lease times or reference counting.

Implementation transparency. Other technologies only provide location transparency or access transparency for component distribution. If location transparency is provided, developers need not know where a remote service is located, but method calls to it are implemented differently than to local objects. If access transparency is provided, location transparency is provided too, but method calls to remote services are implemented in the same way as to local objects. Usually access transparency is provided via proxy objects. However, developers must be aware that consecutive calls may be executed in different threads on the remote side, or that serialized objects may are duplicated multiple times, or that modifications to serialized objects do not get synchronized with the original object. The Plux component model provides implementation transparency for distributed extensions, which implies access transparency and location transparency. To support implementation transparency, Plux offers the following mechanisms: it provides *distributed thread management*, which simulates a single coherent thread that is assembled from multiple distributed threads that are linked together. It provides *reference identity*, which ensures that if an object is transferred to a remote environment multiple times, the remote environment gets the same reference to the remote object each time. Vice versa, if the remote object is transferred back to the original environment, the original environment gets a reference to the original object. Finally, it provides *object data synchronization*, which ensures that if a serialized object is modified in a remote environment, the original object in the original environment is updated too. Due to the support of implementation transparency, remotely plugged extensions can be implemented in exactly the same way as locally plugged extensions.

5.1 Metadata Standard

Extensions for distributed multi-user web applications are implemented in the same way as extensions for single-user desktop applications. Therefore, Plux extensions can be reused in single-user desktop applications as well as in distributed multi-user web applications, regardless of if they are installed as server-side or as client-side extensions. The distribution of extensions is transparent to the developer. Developers just declare extensions by attaching the *Extension* attribute to classes and define the extensions' requirements and provisions using the *Slot* and the *Plug* attributes respectively, which was described in Section 3.1. The metadata standard of the extended component model for the web does not add any further specifications for extension declaration.

5.2 Deployment Standard

Extensions are deployed in plugins, and slot definitions are deployed in contracts. Plugins and contracts are DLL assembly files, which are copied into discovery directories and are discovered by an exchangeable discoverer mechanism (see Section 3.2). In contrast to single-user desktop applications, in multi-user web applications each discovered plugin and contract needs to be assigned to a specific user. For this, the web discovery mechanism supports user-specific and user group-specific plugin repositories. Furthermore, since these repositories can either reside on the web server or on a remote computer, such as the user's client-side computer, the discovery mechanism supports discovering distributed plugins and contracts.

5.2.1 User-specific Repositories

Users can extend their web applications with user-specific extensions, i.e., users can install their individual set of server-side, client-side, and sandbox plugins. For this, a server-side discoverer monitors user-specific plugin directories on the web server and a client-side discoverer monitors a plugin directory for the user on the client-side.

Figure 5.1 shows an example with a server-side repository for two users named *Markus* and *Julia* and an additional client-side repository for each user. Each user has its individual instance of a server-side *Discoverer* extension and an individual instance of a client-side *Discoverer* extension. The server-side *Discoverer* matches usernames with the directory names in the repository to assign directories to users. Within a user directory,

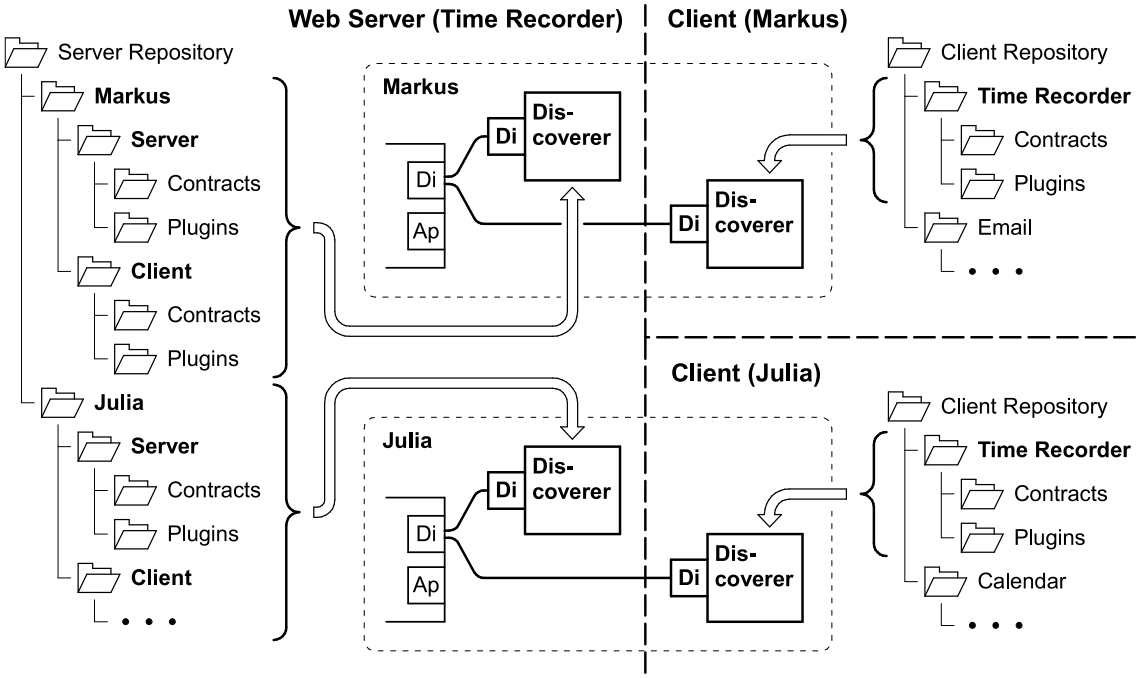


Figure 5.1: Discovery of user-specific server-side, client-side, and sanbox plugins

server-side plugins and sandbox plugins are separated via subdirectories. Server-side plugins are copied into the *Server* directory, while sandbox plugins, which are executed on the client-side, are copied into the *Client* directory. User-specific contracts are installed in the same way as user-specific plugins. Plugins and contracts can optionally be separated in further subdirectories. The *Discoverer* monitors the directories for a specific user and notifies the Plux core when new plugins or contracts are added or when they are removed.

Client-side plugins are discovered with the client-side *Discoverer* extension, which is plugged remotely to the Plux core when the client connects to the server (see Section 5.4.2 Connection Establishment). The client-side discoverer matches the name of the web application with the directory names of the client-side repository, monitors the directory that belongs to the application, and notifies the Plux core when new plugins or contracts are copied into the application directory or when they are removed from there. In Figure 5.1 the name of the application is set to *Time Recorder*. Thus, each client-side discoverer that is connected to this application monitors the *Time Recorder* directory in the client-side repository.

The *Discoverer* extensions in Figure 5.1 are simplified. In the real implementation, each of them has a slot for detector extensions as well as a slot for analyzer extensions, as it was described in Section 3.2.

The directory layout, which is described above, is the default directory layout for discoverer extensions. However, the discoverers can be customized, in order to assign any directory to a user on the server-side, or to assign any directory to an application on the client-side. Also the separation between server-side plugins and sandbox plugins need not be done with subdirectories of a user directory, but can also be done on a different directory level, e.g., on the outermost level of the plugin repository. As the discovery mechanism is implemented with extensions, it can also be replaced with other discoverer implementations, e.g., with a discoverer that retrieves discovery information from a database.

5.2.2 Hierarchical Discovery

Although every user can install his individual set of plugin, the plugins for the base application are usually the same for all users. As it would be inefficient to add all base plugins to all user directories in the repository and because it would be challenging to keep the base plugins for all users consistent on every version update, Plux supports plugin directories that are shared among *user groups*. User groups are organized with a *user store*. Each user group can have members, which can be users or again user groups. Actually, the user store does not distinguish between users and user groups. Therefore, also users can have subordinate members and thus maintain different sets of plugins for a single web application. Vice versa, each user and user group can belong to multiple user groups at the same time.

Figure 5.2 shows an example of a server-side plugin repository with various directories for users and user groups. As user groups (and users) can have multiple members and since members can belong to multiple user groups at the same time, directories for user groups and for users are organized in a flat way in the plugin repository. The user store defines the hierarchy of users and user groups. To keep the example simple, in Figure 5.2 each user directory only contains server-side plugins, but does not distinguish between server-side and sandbox plugins.

The user store defines a user group *Base*, to which all users and user groups belong. *Base* has two subgroups named *Worker* and *Anonymous*. The users *Markus* and *Julia* belong to the group *Worker* and all anonymous users automatically belong to the group *Anonymous*.

Plugins are installed for a user or for a user group by copying them into the right directory. To uninstall them, they are removed from the directory. However, in some situations a plugin that is installed for a certain user group must be made unaccessible for a subgroup of this user group. In such a case, the plugin cannot just be removed from the directory, because this would affect also other users. Thus, besides the plugin repository the discoverer optionally can use a configuration file to adjust the set of detected plugins for a user. With the configuration file, administrators can include or exclude plugins for users or user groups. In the example of Figure 5.2, the plugin *TimeRedorder.dll* is excluded for anonymous users and the plugin *DataStore.dll* is excluded for users in the group *Worker*.

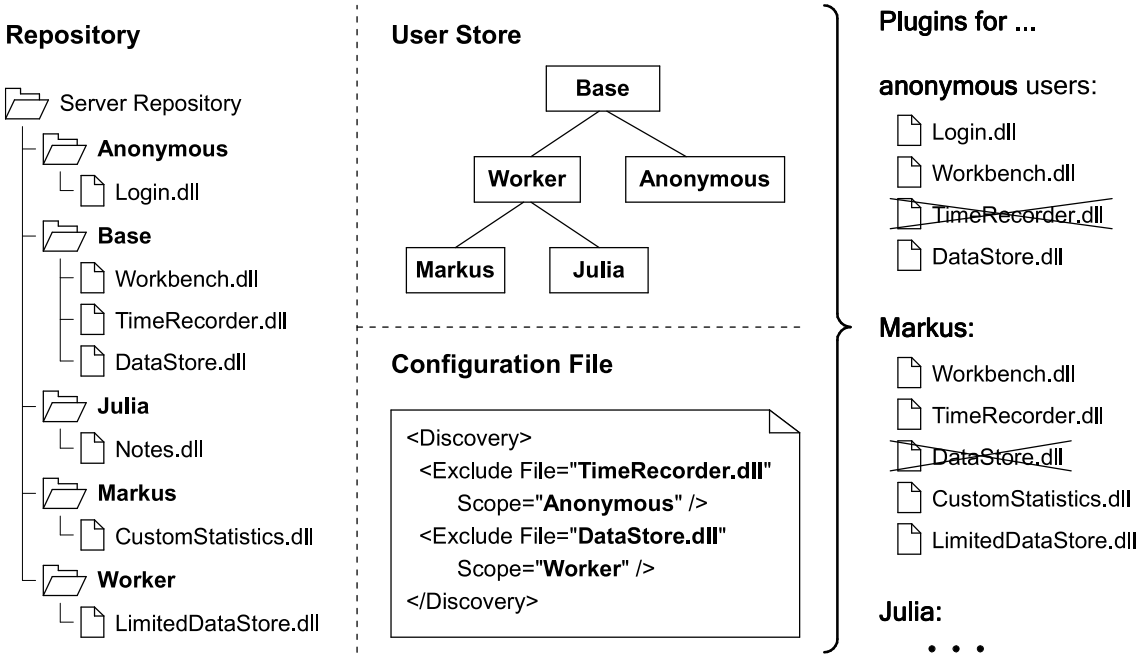


Figure 5.2: Discovery of user-specific and user group-specific plugins using a user store and a configuration file

5.3 Composition Standard

The composition standard specifies how the composer connects extensions and how the composition of an application is maintained by the composition state. Section 3.3 described how the composition state can be used by hosts to retrieve their contributors, which composition operations are used to compose an application, which composition events are raised during the composition process, and how the composer supports automatic composition, programmatic composition, behavior-guided composition, and user-guided composition.

In the extended component model for the web, the composer uses the same composition operations, raises the same composition events and performs the same composition process as specified in the base component model. However, the extended composition standard additionally specifies an *individual composition state* per user, which maintains user-specific extensions in separate user scopes. Furthermore, it specifies a distributed composition process, which is used for automatic composition, programmatic composition, and behavior-guided composition. To provide developers with transparent extension distribution, remote extensions can be accessed in the same way as local extensions via a *distributed composition state*.

5.3.1 Composition State

The composition state stores instances of extensions and their connections. As users are able to plug their user-specific extensions, the composition states of different users vary from each other and need to be maintained individually per user, which is described in the subsection Multi-user Composition State.

Extensions can be distributed over multiple computers. Thus, the composition state needs to be accessible for remote extensions in the same way as it is for local extensions. How the composition state is distributed over multiple computers is described in the subsection Distributed Composition State.

Multi-user Composition State

Plux maintains an individual composition state per user, i.e., every user has its own instances of extensions and its own connections between extensions. This enables users to plug their user-specific extensions, and to unplug extensions without affecting other users. However, providing a multi-user composition state is not just about storing individual instances of extensions and their connections, but also about considering the following issues:

- *User-specific (and group-specific) extensions must only be accessible by those users, who are supposed to access them*. Users must not be able to access user-specific extensions from other users, neither via the Plux infrastructure, nor via any

tricks, such as by instantiating them manually with their constructors or by the use of reflection.

- *User-specific extensions must not lead to conflicts*. If different users install different extensions with equal type names (and equal namespaces), Plux needs to distinguish between them and connect the right extensions for every user.

- *User-specific extensions must not cause errors that affect others*. Enabling user-specific extensions increases the risk of executing untested or buggy code. Nevertheless, in the event of a crash that is caused by a user-specific extension, Plux must ensure that other users are not affected.

In order to provide a solution for these issues, the multi-user composition state allocates separate memory areas for user-specific and for group-specific extensions. As the current implementation of the component model is realized with .NET, the runtime infrastructure uses *AppDomains* [Microsoft, 2013b] to load extensions into different memory areas. Each user-specific (and group-specific) extension is instantiated within a user-specific (or group-specific) AppDomain, i.e., extensions that are installed for the same user (or the same user group) are executed in the same AppDomain, whereas other extensions are executed in different AppDomains.

Figure 5.3 shows an example with two user-specific composition states of the time recorder application for the users *Markus* and *Julia*. Both are members of the user group *Worker*. *Worker* is a subgroup of the user group *Base* (see Figure 5.2 on page 102). Extensions that constitute the base application are installed for all users that are members of the user group *Base* and thus are instantiated in the AppDomain that is called *Base*. Users in the group *Worker* get the group-specific extension *LimitedDataStore* as their data source, which is instantiated in the AppDomain *Worker*. The user *Markus* has not plugged the *Statistics* extension of the base application, but instead uses its user-specific extension *CustomStatistics*, which is instantiated in the AppDomain for all user-specific extensions of *Markus*. Similar, *Julia*'s user-specific extensions *NotesControl* and *Notes* are instantiated in the user-specific AppDomain for *Julia*. As a result, the composition state for a single user is divided into multiple AppDomains, e.g., for the user *Markus* it is divided into the AppDomains *Base*, *Worker*, and *Markus*.

As extensions for different users are loaded into different AppDomains, the type information of extensions is only available for entitled users. Thus, others cannot instantiate them, neither via constructor calls, nor via reflection. Type conflicts between user-specific plugins of different users cannot happen too, because user-specific types are loaded into different AppDomains. Finally, due to user-specific AppDomains, errors that are caused by user-specific extensions do not lead to crashes that affect others; even if extensions crash so badly that the hosting AppDomain crashes, too.

Despite the fact that extensions are instantiated in different AppDomains, multiple instances of the same extension always are created in the same AppDomain, even if

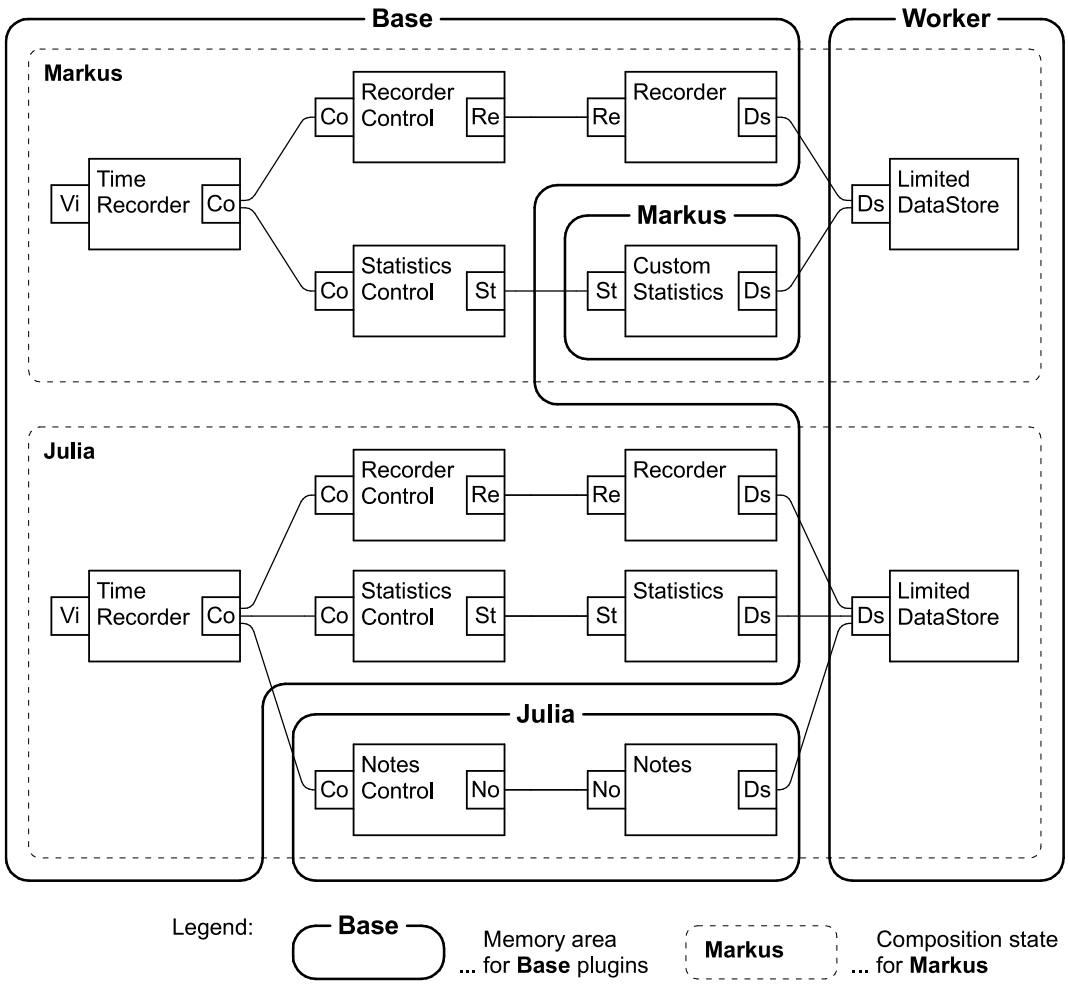


Figure 5.3: Individual composition states per user with extensions that are executed in user-specific and group-specific memory areas

they are created for different users. For example in Figure 5.3 the extension *LimitedDataStore* is instantiated multiple times, one instance for each user that is a member of the user group *Worker*. However, as all instances of the same extension are executed in the same AppDomain, developers still can implement extensions that share data and common resources between different users within the AppDomain.

Unfortunately, if extensions are instantiated in different AppDomains, they cause additional communication overhead, when they call methods across AppDomain boundaries. However, extensions only are divided into separate AppDomains, if user-specific (or group-specific) extensions are plugged to an application, i.e., for a composition without any user-specific extension, there is only one AppDomain and thus there is no extra communication overhead. Even if user-specific extensions are plugged to an application, there is no communication overhead between the extensions of the base application. Only method calls from base extensions to user-specific extensions are affected by the overhead. Also, since extensions that are installed for the

same user group are loaded into the same AppDomain, there is only communication overhead between extensions that are installed for different user groups.

As each plugin is assigned to a certain user (or user group), each plugin is only loaded once, namely in the AppDomain for this user (or user group). However, contracts are handled differently than plugins. As contracts contain slot definitions, which specify the interfaces of slots, the type information of a contract is required by both, the hosts, which use the interface to access their plugged contributors, and the contributors, which implement the interface of a slot definition. In order to enable a host of the base application to open a slot, the contract with the slot definition for this slot must be installed for the base application. However, such a contract is not only loaded in the *Base* AppDomain, for which it is installed, but also in all AppDomains that contain a contributor with a plug for a slot that is defined in this contract. For example, in Figure 5.3, the contract with the slot definition *DataStore* is installed for the user group *Base*, but loaded in all AppDomains, because the extensions *Recorder* and *Statistics* open this slot in the *Base* AppDomain, the user-specific extensions for *Markus* and for *Julia* open this slot in their user-specific AppDomains, and the extension *LimitedDataStore* for the user group *Worker* provides a plug for this slot. The contract that contains the *Statistics* slot definition also needs to be installed for the base application. However, as only the user-specific extension *CustomStatistics* uses this slot (besides the base extensions), only the AppDomains *Base* and *Markus* load this contract.

As user group contracts are also loaded into the AppDomains of user group members, their type information is accessible by user-specific plugins of those members. As a result, user-specific plugins can use the implementation of contracts installed for higher-level user groups. Thus, contracts should only contain interface descriptions, but not sensitive library implementations. Type conflicts can never occur between the contracts of different users. However, they can occur between contracts of a user and contracts of a higher-level user group. In that case, only the slots and plugs can be used that reference the contract that was loaded first. For slots and plugs that cannot be used because of a type conflict, the logger writes a message to the log output. Similar to type conflicts, as user-specific contracts are not loaded in higher-level group-specific AppDomains, they cannot lead to errors that affect other users. However, higher-level group-specific contracts can lead to errors in AppDomains for all members in the user group. Thus, the permission to install group-specific contracts should be granted only to trusted people. This decreases the risk of installing error-prone contracts, as well as the risk of installing contracts that provide sensitive implementations to members of a user group that should not have access to them.

To increase performance on the server, the communication overhead between AppDomains can be avoided by disabling the separation of user-specific plugins into different AppDomains. This may is reasonable if all server-side plugins are maintained by a single administrator. Even if all plugins are executed in the same AppDomain, users can still have their individual sets of plugins as well as their individual

composition states. And even if only the administrator is allowed to install plugins on the web server, users can still plug their individual client-side plugins by themselves.

Distributed Composition State

The distributed composition state enables hosts to access remotely plugged contributors in the same way as they access locally plugged contributors. Furthermore, the composer uses the distributed composition state for the distributed composition process (see Section 5.3.2) that plugs contributors form remote computers to local hosts, and vice versa.

Figure 5.4 shows an example of how a host accesses a remotely plugged contributor by using the composition state. In (1) the extension object of the *TimeRecorder* uses its meta-object to retrieve the meta-object of the plugged contributor *MobileSync* (2). Via the retrieved meta-object the host gets a reference to the contributor's extension object (3) and finally can call the contributor's methods (4).

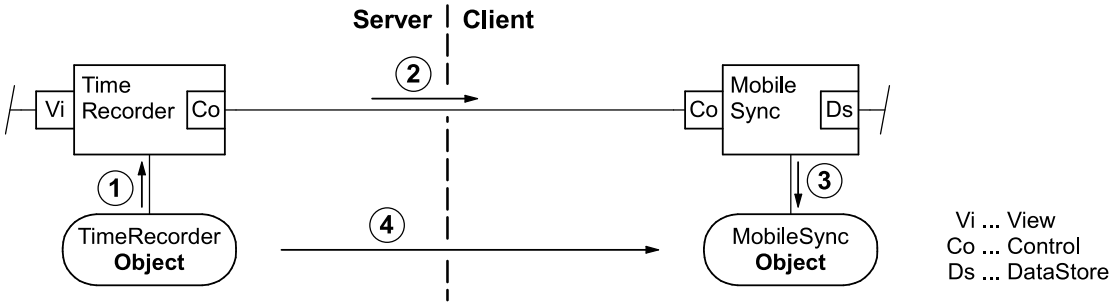


Figure 5.4: Logical view of the distributed composition state with meta-objects and extension objects

However, the distributed composition state in Figure 5.4 cannot be implemented as it is shown, because the extensions *TimeRecorder* and *MobileSync* are instantiated on different computers and the *Control* slot of the of the *TimeRecorder* cannot have a direct reference to the *Control* plug of the *MobileSync* extension. Thus, Figure 5.4 just shows a logical view of the composition state as it appears to users and to developers.

Figure 5.5 on the next page shows how the actual distributed composition state is implemented. On the server-side, the *TimeRecorder* references a copy of the client-side *MobileSync* extension, while the client-side *MobileSync* extension is plugged into a copy of the server-side *TimeRecorder* extension. Both meta-object copies reference a proxy object instead of the original extension object. Thus, the whole composition state is available both on the server and on the client.

When the server-side *TimeRecorder* wants to use the client-side *MobileSync* contributor, it first retrieves the *TimeRecorder* meta-object (1), then the plugged copy of the *MobileSync* meta-object (2) to get a reference to the *MobileSync* proxy object (3). When the *TimeRecorder* calls a method on the *MobileSync* proxy object (4), the proxy uses the *Plux Runtime Coordinator*, which is part of the composition infrastructure and handles

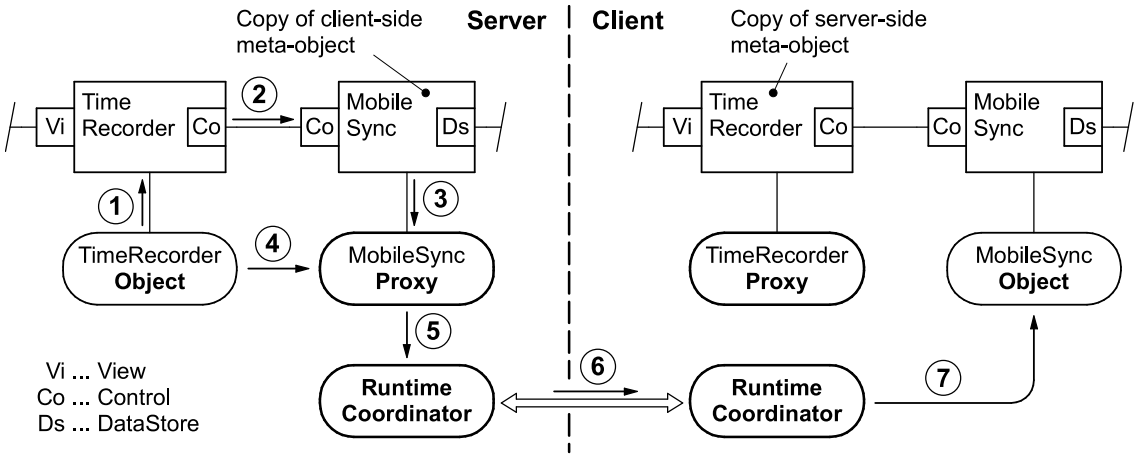


Figure 5.5: Implementation of the distributed composition state with meta-object copies and proxy objects

the communication between distributed extensions, to forward the call to the remote coordinator (6), and finally to call the method on the original extension object of the client-side *MobileSync* contributor (7). The result is sent back in exactly the same way.

5.3.2 Composition Process

The composition process defines how extensions are composed to a coherent application. Section 3.3 described how Plux performs automatic composition, i.e., how the composer retrieves the extensions' requirements and provisions from their metadata and automatically connects matching extensions. Furthermore, it described programmatic composition, behavior-guided composition, and user-guided composition. The composition process for component-based web applications is the same as described for the base component model. However, for distributed multi-user web applications this composition process needs to be performed for multiple users as well as for distributed extensions.

For distributed web applications Plux needs to compose extensions, which are located on different computers. For this, the distributed composition process uses distributed composer instances that are synchronized with a token passing scheme.

Multi-user Composition Process

Plux needs to compose a multi-user web application individually for each user. In order to avoid interference between these composition processes, the composition infrastructure instantiates individual composer instances that perform the multi-user composition in parallel using a dedicated runtime thread per user (see Section 5.4.1 Thread Management).

Distributed Composition Process

The composition standard for the web specifies that extensions can be installed and executed on different computers. Thus, the composition process needs to be distributed, too, in order to instantiate extensions on different computers and connect them to a single coherent web application.

As described in the base component model, the composition process is divided into composition sequences, each of them performing a number of composition operations for a specific extension to make it ready for use, i.e., they activate the extension and fill its slots with contributors (see Section 3.3). The same happens in the distributed composition process. However, the composition sequences are executed by a distributed composer that comprises multiple synchronized composer instances, which are located on different computers.

Plux uses a token passing scheme to coordinate the distributed instances of the composer. The token is passed from one computer to another and indicates the active environment with the composer instance that is allowed to perform the next composition operation. As soon as an operation needs to be executed on a different computer, the token is passed on. The token also coordinates the distributed runtime thread (see Section 5.4.1), which simulates a single coherent thread that is distributed across all connected environments. Only the environment that currently has the token is enabled to execute code in the runtime thread, while all other environments are blocking this thread. As the composition state can only be retrieved from within the runtime thread (see Section 3.4.1), the distributed composition state (see Section 5.3.1) only needs to be kept up to date in the environment that currently has the token. As a result, the composition state is not updated after each performed composition operation in all remote environments, but only when the token is passed to them. The environment with the token always has full knowledge about the current composition state and about all available extensions. Thus, the distributed composer does not communicate with remote environments during the composition process until it forwards the token to the next environment to perform a composition sequence for an extension there or to execute a remote event handler for a raised composition event.

Figure 5.6 on the next page shows how the distributed composition process composes the server-side *TimeRecorder* and remotely plugs the client-side *MobileSync* contributor to the server-side host. In (a) the figure shows the composition state, before the *TimeRecorder* host is activated. The server-side environment currently has the token and thus is active. The active environment is indicated with a solid border and a header above; the runtime thread on the client-side environment is currently blocked, which is indicated with a dotted border and a header caption. As soon as a composition sequence for the *TimeRecorder* is triggered, the server-side composer instance activates the extension and fills its slot. In (b) the composition sequence is finished, i.e., the *TimeRecorder*'s extension object is instantiated and a copy of the meta-object of the client-side *MobileSync* extension is plugged to the *TimeRecorder*. For this, it

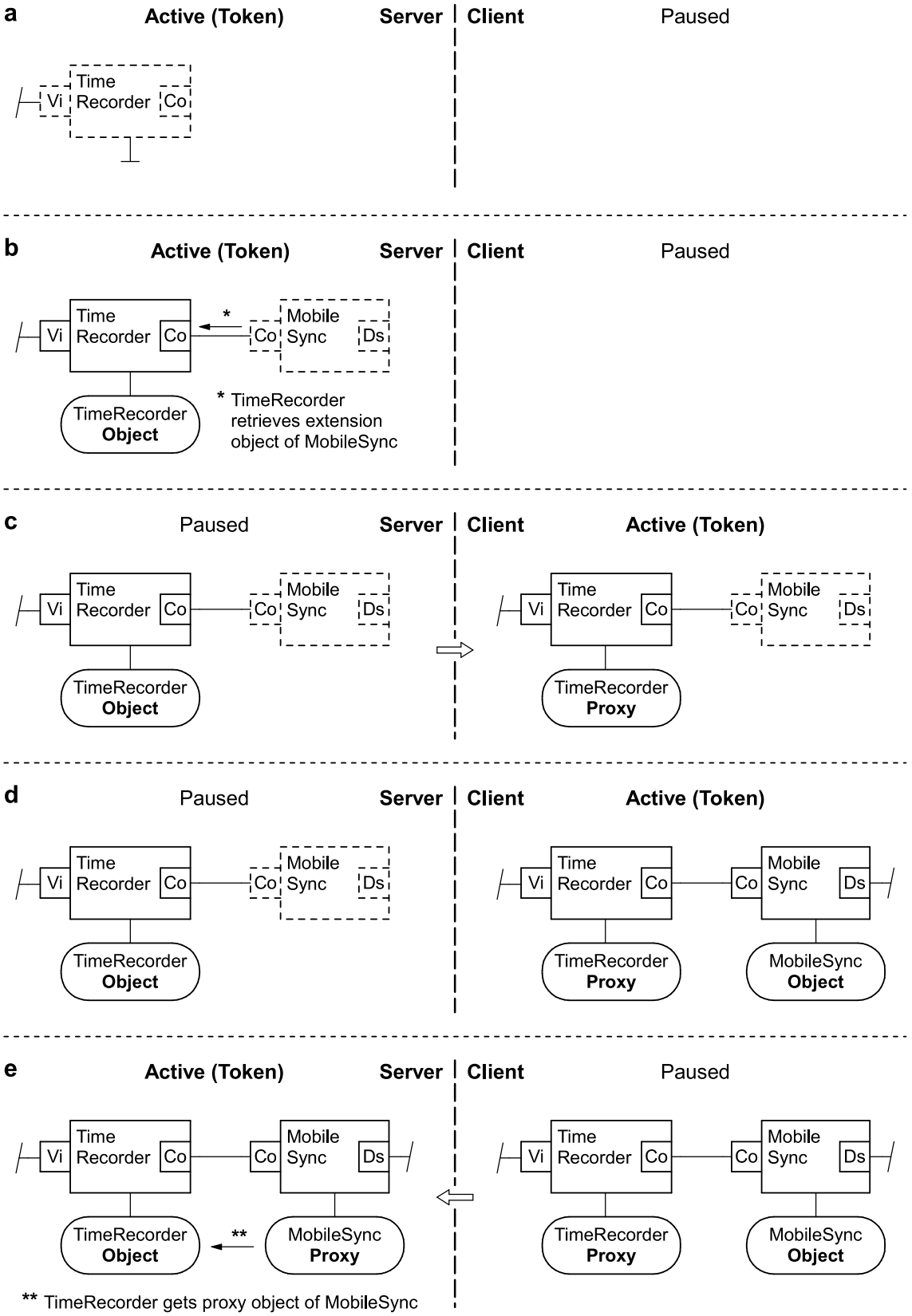


Figure 5.6: Composing distributed extensions using token passing

was not necessary to pass the token to the client-side. Therefore, the composition state on the client-side was not updated yet and still is empty.

When the *TimeRecorder* retrieves the *MobileSync*'s extension object, the composition sequence for *MobileSync* is triggered (see Host-triggered Composition Sequences in Section 3.3.4). As *MobilSync* is a client-side extension, the composition sequence for this extension needs to be performed there. Therefore the composer passes the token to the client-side environment (c).

Every time when the token is passed from one environment to another, the composition state is updated on the target environment that receives the token. Thus, the composition state is transferred from the server-side to the client-side, whereat the *TimeRecorder*'s extension object is replaced by a proxy object there. In (d) the composition process continues with performing the composition sequence for *MobileSync*. The composition sequence instantiates the *MoblieSync*'s extension object and fills the *TimeRecorder*'s slot. When the composition sequence is finished, the token is passed back to the server-side environment (e). There, the server-side composition state is updated with the activated *MobileSync* contributor, which now references a proxy object. Finally the *MobileSync* proxy object is returned to the *TimeRecorder*, which was requested in (b).

Although the distributed composition process is performed on distributed computers, logically it is the same as the local composition process. Thus, all other specified composition concepts, such as composition events, programmatic composition, or behavior-guided composition, can be used in distributed composition in the same way as in local composition.

5.4 Interaction Standard

The interaction standard specifies how extensions communicate with each other and how they exchange data. Plux web applications are used by multiple users at the same time, they are executed in sequential round trips using the request-response method, and they can be distributed across multiple computers. Thus, the interaction standard for the web is extended by additional specifications for thread management and by specifications for communication to provide transparent distribution support. These specifications enable developers to implement distributed extensions in the same way as local extensions without considering threading or communication issues.

To describe the following specifications, this section uses several sequence diagrams. As these sequence diagrams contain additional elements, which are not specified in the *Unified Modeling Language (UML)* [OMG, 2011], Figure 5.7 on the next page explains the elements that are used in the sequence diagrams below. A sequence diagram shows an interaction between objects. Every object is represented by a vertical line, which is called lifeline. The header of the lifeline denotes the name of the object's class. As objects can be instantiated in different environments, sequence diagrams may contain

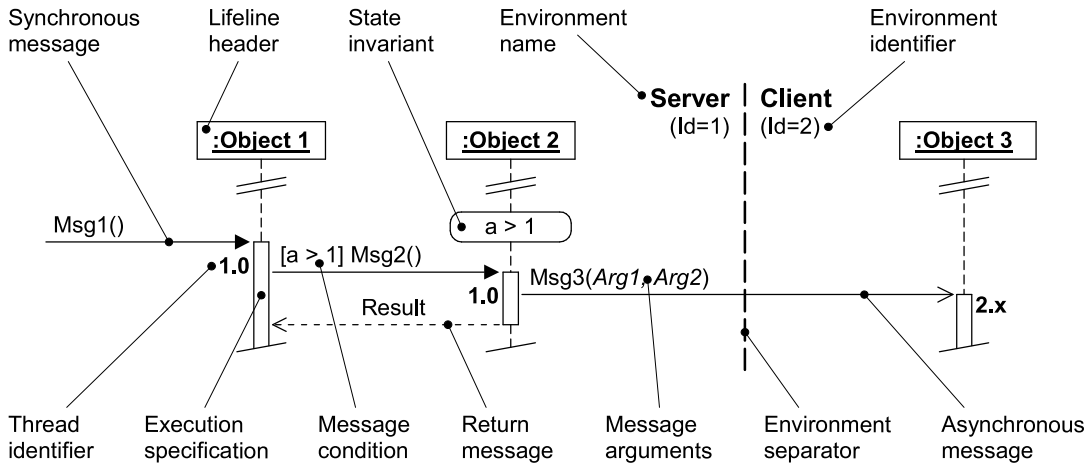


Figure 5.7: Elements of a sequence diagram

multiple environments, which are separated by *Environment separators* that are represented with a dashed bold line. Each environment has an *Environment name* and a unique *Environment identifier*.

Objects communicate with each other using messages. *Synchronous messages* are processed in the same thread as the one in which they were sent. They are represented by a solid line with a filled arrowhead. *Asynchronous messages* are processed in a different thread as the one in which they were sent. They are represented by solid line with an open arrowhead. Each message optionally can have *Message arguments*, which are listed in parentheses after the message name. *Return messages* notify the sender of a *Synchronous message* about the end of its execution and optionally pass a *Result* to sender. *Return messages* are represented by a dashed line with an open arrowhead.

To describe multithreaded processes within a single sequence diagram, threads are marked by *Thread identifiers* that are assigned to each *Execution specification*. *Execution specifications* represent the execution of a message. Every thread identifier consists of two parts that are separated by a dot. The first part is the identifier of the environment in which the thread was started (i.e., the *Environment identifier*). The second part is a consecutive number that is incremented for every new thread. The *Environment identifier* is prepended in thread identifiers in order to allow every environment to create new thread identifiers without synchronizing the consecutive number with other environments. As the runtime thread is always the first thread in the environment *1*, it has the thread identifier *1.0*. In some sequence diagrams the consecutive number of a thread identifier is replaced by a letter, which represents a variable and indicates a random thread, e.g., a thread from a thread pool.

In order to keep sequence diagrams simple, they do not describe alternative traces with combined fragments. However, to indicate a choice of behavior at a certain point, messages can be prefixed with a *Message condition*. *Message conditions* are expressions within brackets that have to evaluate to true in order to start the interaction. The sequence diagrams below only show traces in which the expression is always true,

other traces are shown in separate sequence diagrams in the appendix of the thesis. Finally, *State invariants* indicate a certain state in the process and are represented by a solid oval box containing a description of the current state.

5.4.1 Thread Management

To avoid complicated thread synchronization implementations in every extension, the interaction standard specifies a dedicated runtime thread, which is the only thread that is allowed to execute runtime operations, such as retrieving the composition state or performing composition operations. Furthermore, extensions are only allowed to call methods from other extensions in the runtime thread so that extensions do not have to check whether they were called in the runtime thread, when they want to perform operations on the composition state.

Since web applications are used by multiple users concurrently and since different users should not interfere with each other, the interaction standard for the web specifies a separate runtime thread per user. However, assigning a separate thread to every user would be inefficient, because web applications are not running continuously, but are rather executed in sequential round trips. A user interaction with a web browser triggers a web request, which is sent to the web server. The web application on the web server processes the request, and finally a response is sent back to the browser. After such a round trip the web application is idle, until the next round trip is triggered. If every user would have his own thread, this thread would have to wait for the next request of the same user after every round trip. If threads are shared among different users, they do not have to wait for a certain user, but can process requests from other users in the meantime. Therefore the interaction standard for the web specifies a multi-user runtime thread that is only assigned to a specific user during a round trip, but is released afterwards to reuse it as runtime thread for another user.

Even though extensions can be distributed to different computers, operations on the composition state still need to be executed in a single runtime thread. However, since distributed applications are executed on different computers, they cannot be executed within the same thread. Therefore the interaction standard specifies a distributed runtime thread, which simulates a single coherent thread that comprises multiple threads, which are located on connected environments and which are coordinated with a token passing scheme.

Multi-user Runtime Thread

Web applications are executed in sequential round trips, where each round trip is processed in a different thread that is taken from a common thread pool. As a result, sequential round trips for a certain user are processed in different threads. Nevertheless, the interaction standard specifies only a single runtime thread. In order to reuse threads for different users and to restrict composition operations to the

runtime thread, the runtime thread is taken from a thread pool and needs to be switched for consecutive round trips.

Plux provides a dispatcher, which is used to invoke method calls from any thread in the runtime thread. The dispatcher can be acquired by any thread, but only by one at a time. Acquiring the dispatcher makes the current thread the runtime thread. After finishing a round trip, the dispatcher is released so that other threads can become the runtime thread.

Figure 5.8 shows a sequence diagram that describes how a thread acquires the dispatcher in order to become the runtime thread for processing a web request. The sequence diagram uses state invariants to indicate whether the dispatcher currently is *Acquired* or *Released*. As web applications can also be distributed, the runtime thread can be distributed as well. The distributed runtime thread is coordinated with a token that enables only one environment at a time to execute code in the runtime thread (see Distributed Runtime Thread below). Although the web application in Figure 5.8 is not distributed, the sequence diagram marks the positions with a *Token* state invariant where the token is required for executing code in the runtime thread.

In (1) the Plux core receives a web request in an arbitrary thread *1.x* via its *Process* method. To make the executing thread the runtime thread, the method acquires the dispatcher (2). After the dispatcher is acquired (3), the runtime thread id, which is always *1.0*, now is assigned to the current thread. Next, the dispatcher raises an *Acquired* event, which is handled by the coordinator. The coordinator is responsible for thread management and remote communication. It ensures that it has the token (4) before the *Acquire* method returns to its caller. In other words, the coordinator blocks the runtime thread of the current environment until it gets the token. If the token would be missing in (4), the coordinator would request it from the connected environments and would receive it, as soon as the dispatcher in the environment that holds the token becomes idle.

Even though the core's *Process* method now is already executing in the runtime thread, for error handling reasons, the core ensures that every code that is executed in the runtime thread is initiated from a dispatcher operation. Thus, after the dispatcher is acquired, the core calls the dispatcher's *BeginInvoke* method (5) to asynchronously enqueue a dispatcher operation for the web application's *Process* method to the operation queue of the dispatcher. Each time, when an operation is enqueued, the coordinator handles the dispatcher's *OpEnqueued* event (6). *OpEnqueued* checks whether the current environment has the token. In Figure 5.8 this cannot happen, because the token was already retrieved in (4), and can only be passed to other environments in the runtime thread, which is currently under the control of the core's *Process* method.

In (7) the core calls the dispatcher's *Run* method to start executing the enqueued dispatcher operation. The argument of *Run* specifies whether the dispatcher should wait when its operation queue is empty. Passing the argument value *false* causes the *Run* method to return to the caller as soon as the dispatcher is idle. The dispatcher

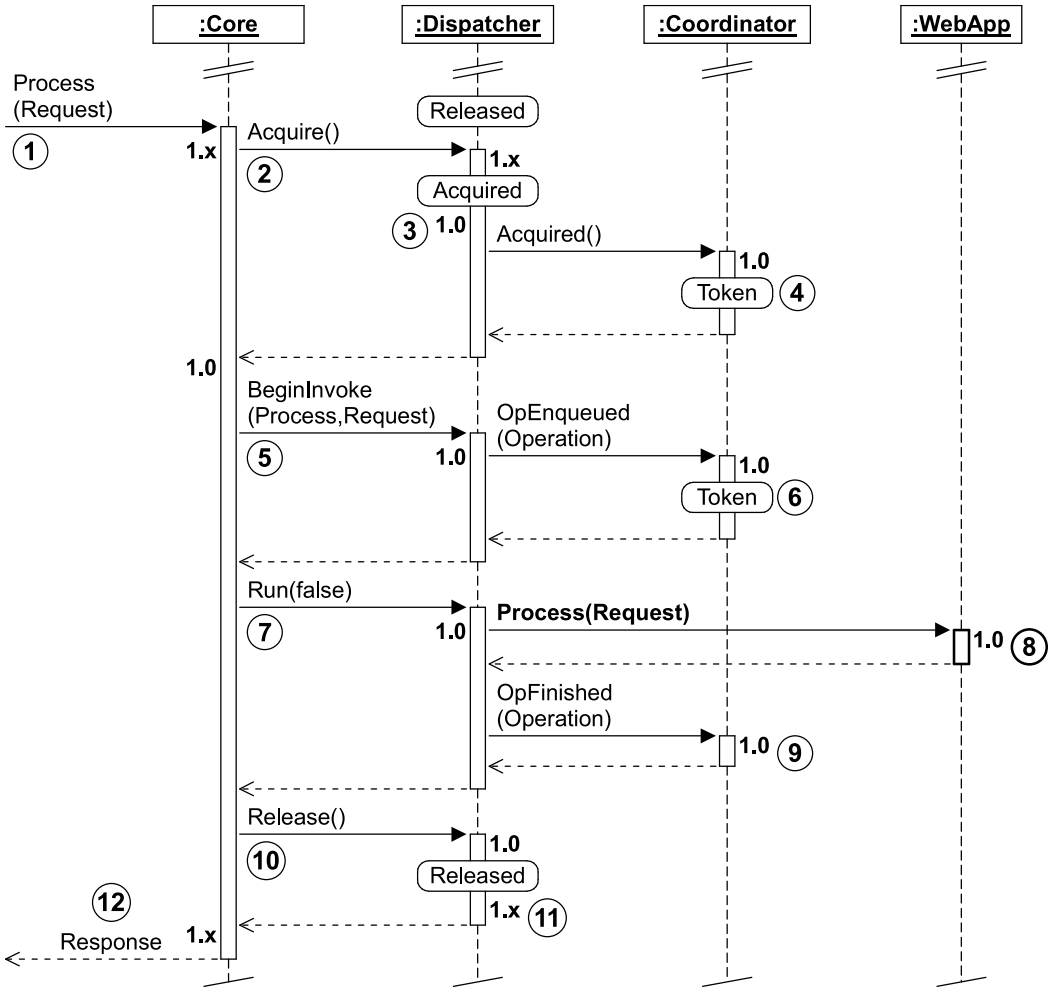


Figure 5.8: Acquiring the dispatcher to process a web request in the runtime thread

operation in the queue invokes the *Process* method of the web application (8). After executing the *Process* operation is finished, the dispatcher notifies the coordinator (9) to check whether the next dispatcher operation was enqueued from a remote environment. If so, the coordinator would pass the token to that environment and continue execution there. As the dispatcher's *Run* method does not return to the caller until all dispatcher operations are finished, processing a single web request can include the invocation of further operations during one round trip.

When the dispatcher is finished, the core releases the dispatcher (10) and the current runtime thread becomes the original thread *1.x* again (11). Finally, in (12) the response for the request is returned to the caller of the core's *Process* method.

The sequence diagram in Figure 5.8 shows the idea of acquiring and releasing the dispatcher to swap the runtime thread. However, this sequence diagram is simplified; the full sequence diagram can be found in Appendix B: Runtime Procedures.

Distributed Runtime Thread

In order to be able to distribute web applications across multiple computers, the interaction standard specifies a distributed runtime thread. The distributed runtime thread consists of multiple distributed threads that are synchronized with each other by the use of a token. It simulates a single coherent thread where each connected environment has one thread that constitutes the local runtime thread. Only the local runtime thread of the environment that currently has the token is allowed to execute code, while other local runtime threads have to wait until the token is passed to their environment.

Figure 5.9 shows a sequence diagram that describes how an operation, which involves two environments (e.g., a distributed method call), is executed in the runtime thread. At the beginning the *Server* environment has the token (1) and thus is allowed to execute code in the runtime thread. As the *Client* currently does not have the token (2), the *Client* coordinator called *Wait* (3) in its local runtime thread, after it has sent the token to another environment. *Wait* blocks the thread until it gets a signal to resume.

In (4) a proxy object's *Do* method is called in the runtime thread on the *Server*. As the target object lives on the *Client*, the operation needs to be executed there. The proxy forwards the call to the coordinator on the *Server* (5), which sends a *Call* message with

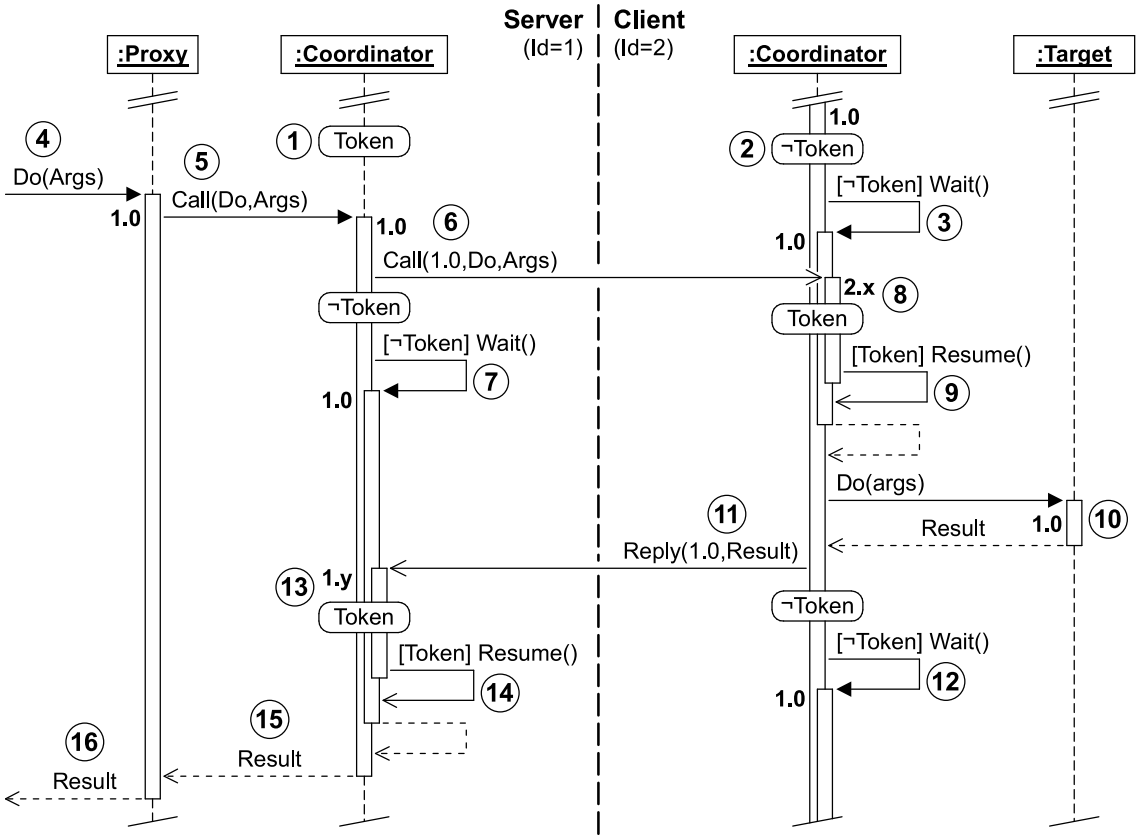


Figure 5.9: Executing a remote operation in the distributed runtime thread

the thread id, the token, the operation to be called, and the operation's arguments to the *Client* (6). After the message is sent, the *Server* does not have the token anymore and thus the coordinator blocks the runtime thread by calling *Wait* (7).

In the meantime the *Client* coordinator receives the message in a background thread for communication (8). Communication threads vary for each received message, which is indicated by the variable *x* in the thread id. As the message contains the token for the runtime thread, the *Client* resumes the local runtime thread (9) to execute the received operation by calling the method *Do* on the target object (10). After the operation is finished, the coordinator sends a *Reply* message with the runtime thread id and the *Result* of the operation to the *Server* (11). As the token is sent with the *Reply* message, the runtime thread on the *Client* is blocked again (12). When the *Server* receives the message with the token in the communication thread *1.y* (13), the coordinator resumes the runtime thread (14) and returns the received result to the proxy (15). Finally, the proxy returns the result to the caller of the method *Do* (16).

5.4.2 Connection Establishment

Web applications are executed on a web server and are accessed via a web browser on a client-side computer. In order to allow a server-side web application to integrate client-side plugins, both, the web server and the web browser, must use a Plux runtime infrastructure that is connected to each other. This section describes how the server-side runtime is started by the web server, and how the browser connects the client-side runtime to the server. After the server-side runtime and the client-side runtime are connected, they remain connected until the user terminates the session or the session is terminated because of a timeout.

When a web application is accessed via a web browser, the web server renders a web page and replies it to the web browser, which displays the web page to the user. If such a web page contains a Plux web control (see Appendix A: Hosting Plux Web Applications), the web control starts the Plux runtime on the server, when it is rendered for the first time. Furthermore, the web control inserts the output of the Plux web application, which is plugged into the *Application* slot of the Plux *Core* extension, into the web page on every round trip. If the client-side web browser has installed a Plux browser plugin, the web browser detects that the received web page contains content of a Plux web application and connects a client-side runtime to the server-side runtime, if the user has installed client-side plugins for the web application.

Figure 5.10 on the next page shows how the runtime infrastructure of a distributed web application is assembled when the web application is accessed by a user for the first time. In (1) the user triggers the start of the web application by entering a web address into his web browser on the *Client*. The web browser sends a request to the web server. During rendering the web page on the *Server* a Plux web control creates the server-side runtime, if it is not already created. (3). While starting the runtime (4), it discovers the server-side plugins for the current user and composes the web

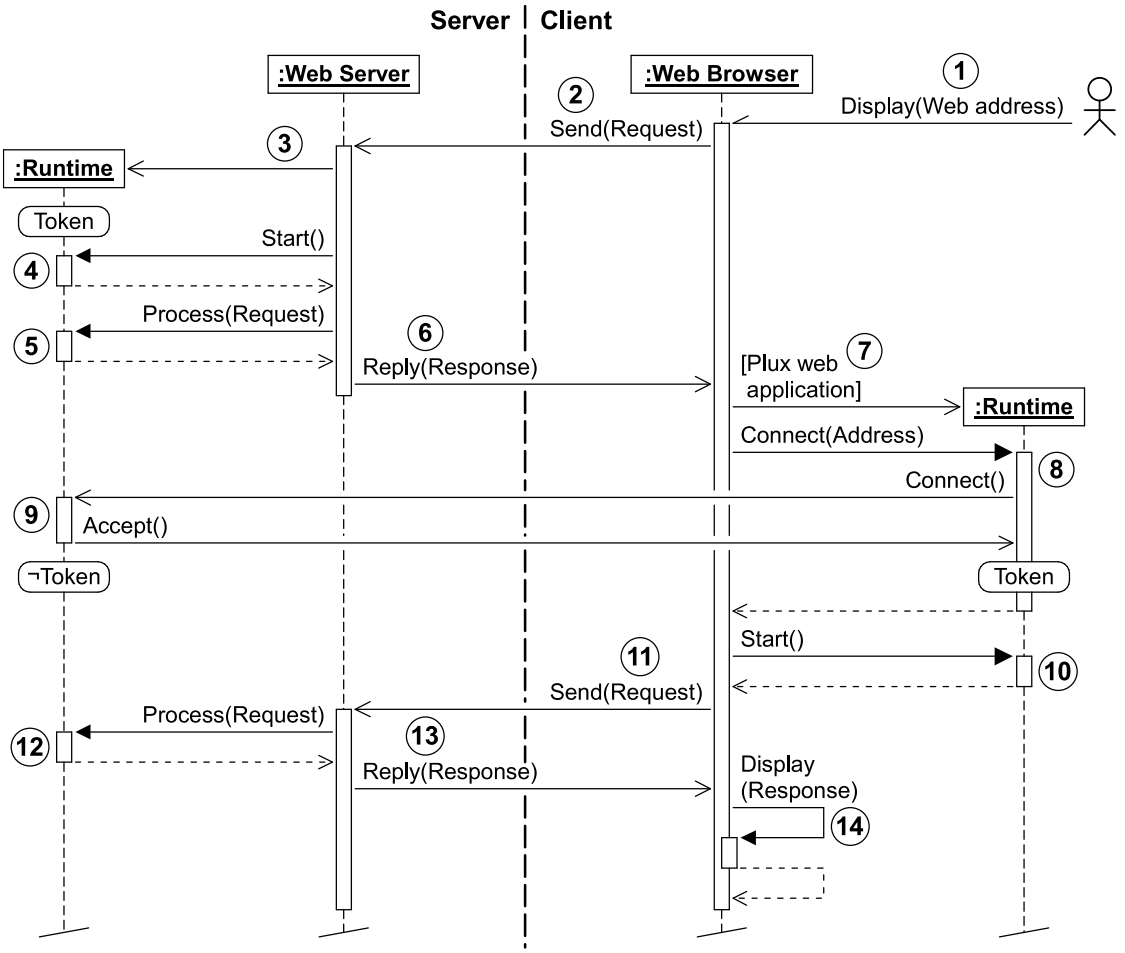


Figure 5.10: Assembling a distributed runtime infrastucture

application. Afterwards, the web application processes the web request (5) and inserts the result into a response message, which finally is replied to the web browser (6).

The web browser receives the response and detects that it was rendered by a Plux web application. As the client-side computer has installed plugins for the web application, the browser does not yet display the web page, but rather instantiates the client-side runtime (7) and calls *Connect* with the address of the server-side runtime as an argument (8). The address of the server-side runtime is retrieved from the response message of the web server (see below). The client-side runtime sends a *Connect* message to the server-side runtime, which replies with an *Accept* message (9). After connecting the runtimes, the token is passed from the server-side to the client-side runtime, which is now prepared to be started in the runtime thread. By starting the client-side runtime (10) the client-side extensions get plugged to the web application. As the web application has been changed, the web page needs to be rendered again. Thus, the browser resends the web request to the web server again and the web application processes it for a second time. Finally, the web server replies the new response to the web browser, which now displays the web page to the user. As the

client-side runtime now is connected to the server, subsequent requests only have to be processed once.

In order to allow the client-side browser plugin to detect that a web server runs a Plux web application, the Plux web control renders a *script* tag with a custom MIME type [Freed and Borenstein, 1996] for Plux applications. Furthermore, the *script* tag contains the connection string that is used to connect the client-side runtime to the server.

Listing 5.1 shows the output of a Plux web control within a *form* tag that was rendered by an ASP.NET web page. The web control inserts the *script* tag with the MIME type *application/plux*, which indicates that the content of the following *div* tag is rendered by a Plux web application. Additionally, the source attribute of the *script* tag provides the connection address *plux://timerecorder.jku.at:25400?session=2658135&app=TimeRecorder*, which is used by the browser plugin to connect the client-side and the server-side runtime. The address is a *Unified Resource Identifier (URI)* [Berners-Lee et al., 2005] that starts with the scheme *plux*, followed by the address at which the server-side runtime is listening for new connections. The query contains the session id and the application name, which are used by the connection listener to assign incoming connections to the correct runtime instance on the *Server*. After the *script* tag, the web control appends the output of the web application, which is wrapped within a *div* tag with an *id* that comprises the keyword *plux* and the application name, which is *TimeRecorder*.

```
<form method="post" action="TimeRecorder.aspx" id="form1">
  ...
  <script type="application/plux"
      src="plux://timerecorder.jku.at:25400
           ?session=2658135&app=TimeRecorder" />
  <div id="plux:TimeRecorder">
    <!-- Plux Web Application -->
    ...
  </div>
  ...
</form>
```

Listing 5.1: The output of a rendered web application including a script tag with a custom MIME type for Plux applications and a source attribute that provides an address for connecting a remote runtime node

Similar to client-side runtimes, if the server discovers sandbox plugins, which need to be executed in a Silverlight runtime on the client, the web control renders a further *script* tag (again containing the connection address) that starts the Silverlight runtime. The Silverlight runtime connects to the server-side runtime. Then the server-side runtime transfers the sandbox plugins to the client-side and the sandbox extensions are remotely plugged to the distributed web application.

5.4.3 Communication Operations

In Plux, the runtime coordinator is responsible for thread management and communication between connected runtime nodes. For this, it uses a set of communication operations, which are described in this section. The coordinator uses channels to transfer messages from one environment to another. Channels provide an abstraction from the transport layer and are exchangeable. The current implementation of the component model uses channels that are connected via sockets and communicate via the *Transmission Control Protocol* (TCP) [Postel, 1981].

Plux distinguishes between target operations that are sent to a specific target environment to be executed there and token operations that can be executed on any environment that currently has the token. An environment without the token does not know which environment currently has the token. Thus, token operations are sent to one environment after another, until the environment with the token receives them. Every environment that forwards a token operation to another environment temporarily stores that operation until it either receives a reply message indicating that the operation was executed or until it receives the token to execute the operation itself. This is necessary because the token might be on the way to an environment, which just forwarded the token operation. In that case, token operations would not be executed by any environment if the operation would not be temporarily stored. Figure 5.11 shows an example.

In the example of Figure 5.11 a runtime on an *Application Server* and a runtime on a *Client* computer are connected to a runtime on a *Web Server*. The coordinator of the *Application Server* needs to execute a token operation *TokenOp* in the thread *2.1* (1). As the environment currently does not have the token, it forwards the operation to the *Web Server* (2) and stores the operation (3) before it waits for a reply (4).

When the Web Server receives the token operation, it does not have the token, too, and thus forwards the operation to the *Client* (5), and stores the operation (6). At the same time, the *Client*, which currently has the token, is executing code in the runtime thread *1.0* and sends a target operation to the *Web Server* (7). As the target operation is sent in the runtime thread, the token is passed to the *Web Server,* too. Thus, the *Client* does not have the token anymore (8) and blocks the runtime thread (9). Now the *Client* receives the token operation (10). As it does not have the token anymore and as it is not connected to any environment that has not yet received the token operation, it replies a message to the *Web Server* that the token operation is still pending (11). In order that an environment knows to which environments the token operation was already sent, the coordinator appends the set of visited environments to messages for token operations.

In the meantime the *Web Server* received the target operation with the token (12), which was sent by the *Client*, in a communication thread *1.y*. As the *Web Server* has stored the pending token operation, it can now start executing it in the thread *2.1* (13), which is the thread in which the operation was sent from the *Application Server*. The communication thread *1.y* has to wait (14) until the token operation is finished, before

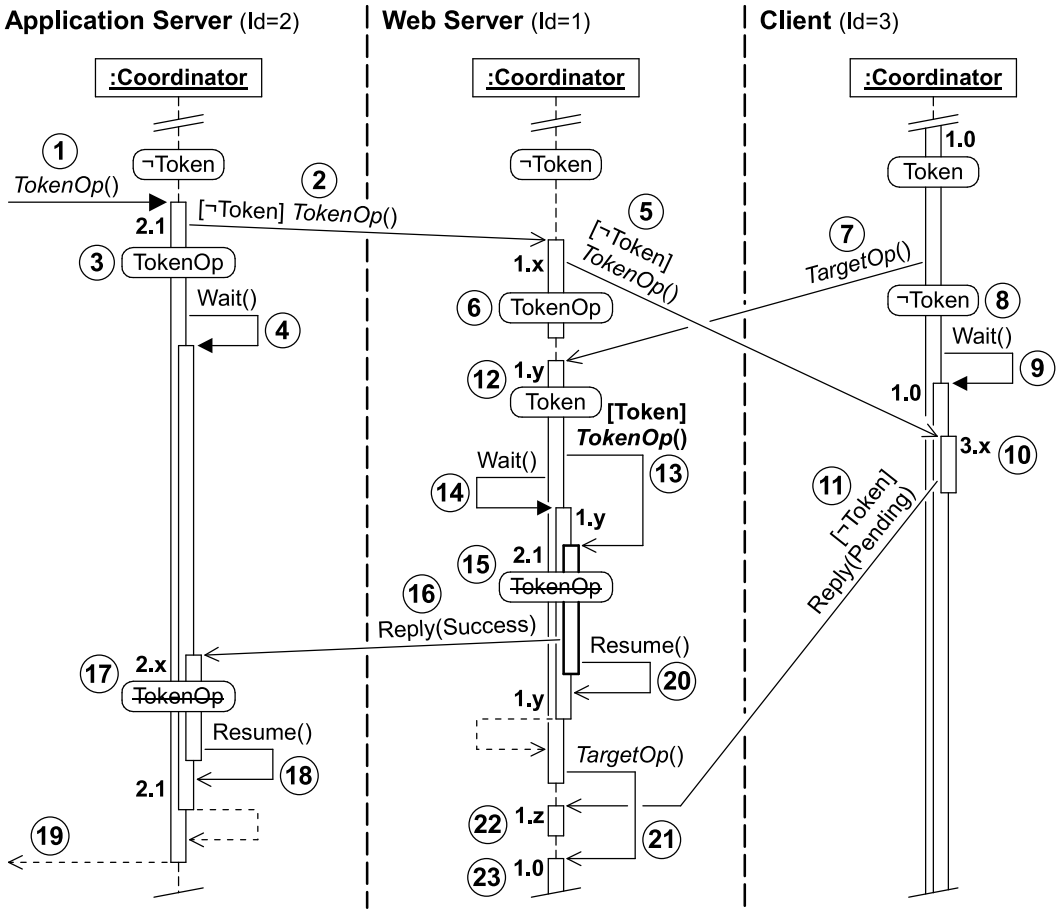


Figure 5.11: Executing a token operation on the environment with the token

it is allowed to start executing the target operation. If it would not wait and start the target operation in parallel, the token might be sent to another environment while executing the target operation. In this case, the token operation would be executed twice. On executing the token operation, the coordinator removes it from the pending operations store (15) and sends a *Reply* message to the *Application Server* that the token operation has finished executing successfully (16).

When the *Application Server* receives the *Reply* message, it removes the finished token operation from its operation store (17) and resumes the waiting thread *2.1* (18). If the *Application Server* would receive a *Reply* message that reports that the operation is still pending, it would not remove the operation from the store, but would forward it to the next connected environment instead.

After the *Web Server* has sent the *Reply* message, it resumes the communication thread *1.y* (20) to start the pending target operation in the runtime thread *1.0* (21). At this point, processing the *TargetOp* message has finished and the coordinator can process the next message, which is the *Reply* message from the *Client* that reports that the token operation is still pending (22). Becauses the operation is already executed and removed from the pending operations store, the *Web Server* ignores the message. If the

token operation would still be stored, the *Web Server* would send the operation to the next environment that did not yet receive the operation. If there would be no environment left, it would forward the *Reply* message with the argument *Pending* to the *Application Server*. After executing the target operation (23), the result and the token will be replied back to the *Client* computer (not shown).

Each operation, whether it is a target operation or token operation, is answered by a *Reply* message, which returns a result to the sender. The reply contains an operation status as well as result arguments. The status can be *Success*, *Pending*, or *Exception*, the result arguments depend on the status:

- *Success* is replied if the execution of the operation was completed successfully. In this case, the result arguments are operation-specific and are described in the subsections for the different operations below.

- *Pending* is replied if the operation was not executed, because the target environment for an operation was not found. This can happen for token operations, if the token was not in any of the environments, to which the operation was already sent. If an environment receives a reply with the status *Pending*, it forwards the operation to the next unvisited environment. The result argument is the set of already visited environments.

- *Exception* is replied if the execution of an operation raised an exception. The result arguments contain the exception object, which is then raised by the environment that originally sent the operation.

In order to let the receiver of an operation know in which thread an operation needs to be executed, each operation contains the thread id of the thread in which the operation was sent. Vice versa, in order to let the receiver of a reply know which thread should be continued, each reply contains the thread id, too.

The following subsections list the different communication operations. Each subsection describes the purpose of the operation, which parameters it needs, and which results it returns by the reply message.

Call

Call is a target operation, which is used to instruct the coordinator of a specific target environment to call a method of a given target object. The arguments comprise an object identifier, which references the target object on the target environment (see Section 5.4.6 Object Reference Identity), the method to be called, and an array of arguments that are passed to the method, when it is called. The reply message contains the return value of the executed method, or no result arguments, if the method does not have a return type.

The *Call* operation is used by proxy objects to forward method calls from remote environments to the original object on the original environment. Furthermore, the *Call* operation is used by the composer for the distributed composition process. For example, the composer starts a composition sequence for a remote extension by calling the *Activate* composition operation on the composer of the remote environment with the extension as an argument.

Invoke

Invoke is a token operation that is used to send a dispatcher operation to the environment which currently has the token. As only the dispatcher in the environment with the token has a valid state of its operation queue, it is the only environment that is allowed to enqueue new dispatcher operations. The argument of *Invoke* is the dispatcher operation to be enqueued. The reply only comprises the operation status, but has no additional result arguments.

The *Invoke* operation is used by the coordinator, when it handles the dispatcher's *OpEnqueued* event. If the environment, in which the operation is invoked, does not have the token, the operation is sent to the environment with the token. As soon as the operation is about to be executed, the token is passed to the original environment and the dispatcher operation is executed there.

GetToken

GetToken is a token operation, which is used to request the token in order to continue execution in the runtime thread. It marks the token to be sent to the sender of *GetToken*, as soon as the runtime thread is idle. *GetToken* has no arguments and no return value.

The *GetToken* operation is used by the coordinator, when it handles the dispatcher's *Acquired* event. If the coordinator does not have the token, it blocks the runtime thread and sends the token request to the environment with the token. When the sender receives the token, it resumes the runtime thread to continue executing.

SetToken

SetToken is a target operation, which is used to pass the token to a specific target environment in order to continue executing the runtime thread there. This operation can only be sent from the environment that has the token. After sending the operation, the environment does not have the token anymore and thus blocks the local runtime thread. The argument of *SetToken* is the token. The reply has no operation-specific result arguments.

The *SetToken* operation is used to send the token to another environment when the runtime thread becomes idle and the token was requested by this other environment before. It is also used when an upcoming dispatcher operation was enqueued from a remote environment and needs to be executed there.

Disconnect

Disconnect is a target operation, which disconnects the sender environment from the target environment. As disconnecting an environment changes the topology of the distributed runtime, this operation only can be performed in the runtime thread when it is idle. Otherwise, disconnecting an environment would disturb the execution of a communication operation. *Disconnect* has an argument that specifies whether the receiver should keep the token. When a runtime is disconnecting its last connection, it instructs the receiver not to return the token in the *Reply* message, even if it is sent in the runtime thread. The result arguments of the reply either contain the token, or not.

The *Disconnect* operation is used by the coordinator, after a runtime closed down. After the operation was sent to the last connected environment, the token stays on this remote environment and the local coordinator releases the dispatcher. Finally the former local runtime thread terminates, too.

5.4.4 Object Transmission Mode

Objects that are transferred from one environment to another exist on multiple environments and are called distributed objects. For distributed objects the remote environment either creates a copy of original object, which is then called a *serialized object*, or it creates a proxy object for the original object, which is then called a *remote object*. This section describes the difference between serialized objects and remote objects and which of them is used for which data type.

Serialized Objects

Serialized objects are completely transferred to a remote environment, i.e., the data of all their fields gets serialized, transferred to the remote environment, and deserialized there. As the remote environment has a copy of the original object, method calls on serialized objects are executed locally. Thus, the call and its arguments need not be forwarded to the original environment.

The advantage of serialized objects is that they do not cause any communication overhead for method calls. However, the disadvantages are that they cause more communication overhead when they are transferred themselves, that the implementation type of a serialized object has to be available on the remote environment in order to be able to instantiate the object, and that modifications on the serialized object have to be synchronized with the original object in order to keep the objects consistent.

Objects are transferred as serialized objects if they have one of the following data types:

- *Value types.* Value types are integer types, floating-point types, the boolean type, user-defined structs, and enumeration types.

- *Types that are decorated with the Serializable attribute.* Classes that are decorated with the *Serializable* attribute are serialized and deserialized using reflection. Listing 5.2 shows an example of a *TimeRecord* class that is decorated with the *Serializable* attribute.

```
[Serializable]
class TimeRecord {
  DateTime start;
  DateTime end;
}
```

Listing 5.2: *TimeRecord* class decorated with the *Serializable* attribute

- *Types that implement the interface ISerializable.* Classes that implement the *ISerializabe* interface are serialized using the interface method *GetObjectData*. The serializer calls this method and passes a *SerilizationInfo* object as an argument. The object writes its data into the *SerilizationInfo* object, which then gets serialized. To deserialize such an object on the remote environment, the serializer uses a specific constructor that gets the *SerilizationInfo* object as an argument. Listing 5.3 shows an example of a *TimeRecord* class that implements the *ISerializable* interface.

```
class TimeRecord : ISerializable {
  DateTime start;
  DateTime end;
  TimeRecord(SerilizationInfo info) { ... }
  void GetObjectData(SerilizationInfo info) { ... }
}
```

Listing 5.3: *TimeRecord* class that implements the *ISerializable* interface

Remote Objects

Remote objects are not copied to a remote environment. Thus, their data is not transferred, but only their data type, which describes the object's interface. The remote environment creates a proxy object, which forwards method calls to the original object. As method calls are eventually executed in the original environment, arguments need to be transferred on each method call.

The advantages of remote objects are that they cause low communication overhead on object transfer, that only the interface types need to be available on the remote environment needs, and that there is no need for synchronization if these objects get modified, because the state of remote objects is only stored in the original object. However, the disadvantages of remote objects are that they cause communication overhead on every method call and that they are difficult to debug.

Objects are transferred as remote objects if they have one of the following data types:

- *Reference types that are not marked with the Serializable attribute and do not implement the ISerializable interface.* Thus, every object of a reference type that is not transferred as a serialized object is transferred as a remote object.

- *Reference types that cannot be created in the remote environment.* Even if a class is implemented to be serializable, the serializer creates a remote object if the implementation type is missing on the remote environment.

Customized Object Transmission

For some objects, such as objects of the types *Assembly*, *Type, MethodInfo*, or *Delegate*, it is neither possible to transfer them as serialized objects nor as remote objects. For those objects, the serializer uses a custom transmission mode, which is implemented by custom formatters (see Section 5.4.5) that are registered for a set of specific data types.

By using custom formatters, applications can be made interoperable between different technologies. For example, a *Type* object or a *List* object in a .NET environment can be transferred to a *Class* object or to an *ArrayList* object in a Java environment. Even though a Java implementation of the Plux component model is part of our future work, objects of several .NET library classes (e.g. *List*, *Dictionary*, or *HashSet*) are transferred by custom formatters because they can be serialized in a more compact way as if they would just serialize their fields via reflection.

Furthermore, since it depends on an object's data type, if the object is transferred as a serialized object or as a remote object, the implementer of the data type determines the transmission mode. However, in some situations a developer wants to specify the transmission mode of *existing* data types. In this case, he can use custom formatters that transfer distributed objects in the desired way.

5.4.5 Object Transmission Format

The interaction standard specifies a technology-independent object transmission format that can be written in different languages. The current implementation of the Plux component model provides writer and reader implementations for XML and for the more compact Plux transmission language, which is aligned to the transmission format. This section describes the structure of the transmission format, the Plux transmission language, and the format for common data types.

The transmission format comprises the entities *Elements*, *Properties*, *Collections*, and *Literals*:

- *Elements* map to objects, types, fields, and to any further entities that are used to describe the data. Elements comprise an element kind, an optional identifier and a set of properties.

- *Properties* are key/value pairs, where the value can be an element, a collection, or a literal.
- *Collections* are homogenous sequences containing a number of elements, a number of collections, or a number of literals. Collections can be empty.
- *Literals* represent simple values such as numbers, booleans, or strings.

The Plux serializer uses the top-level element kinds *Object*, *Type*, *Method*, *Assembly*, *Array*, *Delegate*, *Null*, *Proxy*, *Destroyed*, and *Custom*. These elements are serialized and deserialized by appropriate formatters. Each formatter can define custom element kinds for its sub-elements. As the output of each custom formatter is wrapped within a *Custom* element, they need not care about name clashes of elements with those of other formatters. Property keys only need to be unique within the same element, which is always formatted by the same formatter.

In order to allow two connected runtimes to communicate with each other, they have to use the same formatters. Thus, at connection time the connecting runtime node sends a list of its registered formatters in the connect message. The runtime that receives the connect message compares the formatter list with its registered formatters. Only if they match, the receiver replies with an accept message, otherwise it replies with a reject message containing a list of missing formatters.

Plux Transmission Language

The Plux transmission language is designed to represent the entities described above in a compact way. It is text-based and readable by humans. Listing 5.4 shows its grammar. The basic elements of the language are *kind*, *identifier*, *key*, and *value*, which can be any sequence of characters. Special characters like *(*, *)*, *[*, *]*, *{*, *}*, *<*, *>*, *"*, ** have to be escaped with an backslash (\). An *element* starts with its element *kind* followed by an optional *identifier* and an optional property list. The *identifier* is enclosed in angle brackets; the property list is enclosed in braces and contains one or multiple properties. A *property* starts with its property *key*, followed by its value that is enclosed in parentheses. A property value can be an *element*, a *collection*, or a *literal*. A *collection* is enclosed in brackets and contains a list of *elements*, a list of *collections*, or a list of *literals*. A *literal* is a string that is enclosed in double quotes.

```
Element    = kind ['<' identifier '>'] ['{' Property {Property} '}'].
Property   = key '(' ( Element | Collection | Literal ) ')'.
Collection = '[' ( {Element} | {Collection} | {Literal} ) ']'.
Literal    = '"' value '"'.
```

Listing 5.4: Grammar of the Plux transmission language

The following subsection uses the Plux transmission language to describe the output of formatters for common data types. The formatters follow a naming convention according to which element kinds start with uppercase letters while property keys start with lowercase letters. Since there are only a few top-level element kinds and since property names only need to be unique within the same elements, the formatters use shortcuts for element kinds and property keys.

Format Example

Listing 5.5 shows a part of the implementation of the *Call* operation and how it is instantiated and assigned to an *op* variable. Figure 5.12 shows how this object is serialized by the object formatter. The class *Call* inherits the base class *Operation* and has the field *method* of type *String* and the field *args* of type *Object[]*. Further fields are not shown. The base class *Operation* implements the interface *IOperation* and has the field *threadId* of type *String*.

```
class Call : Operation {
  String method;
  Object[] args;
  ...
}
```

```
class Operation : IOperation {
  String threadId;
  ...
}
```

```
IOperation op = new Call("SetAge", new object[] { 23 }, ...);
```

Listing 5.5: Implementation and instantiation of a *Call* operation

Figure 5.12 describes the format of the serialized operation object from Listing 5.5. The object was formatted with the object formatter, which is used for objects without a custom formatter. The object's type was formatted with the type formatter. The object formatter generates an *Object* element with an object identifier *1.1* that is unique for every application run (see Section 5.4.6 Object Reference Identity). The *Object* element has a *type* property (for the object's type) containing a *Type* element as well as a *levels* property containing a collection of *Level* elements for all inheritance levels with the field values for the respective inheritance level.

As types are objects too, the *Type* element has a unique object identifier *1.2*. *Type* elements have the properties *assembly*, *name*, *generic types*, and *interfaces*. The *assembly* property contains an *Assembly* element, which is formatted by the assembly formatter. The *name* property contains the type name, which is the literal *Call*. Only if a type is generic, the formatter generates a *generic types* property that contains a collection of *Type* elements that describe the generic type arguments. If a type implements one or more interfaces the formatter generates the *interfaces* property that contains the interface types. Interface types are added in order to allow the serializer to create a proxy that implements these interfaces, if the implementation type cannot be loaded on the target environment.

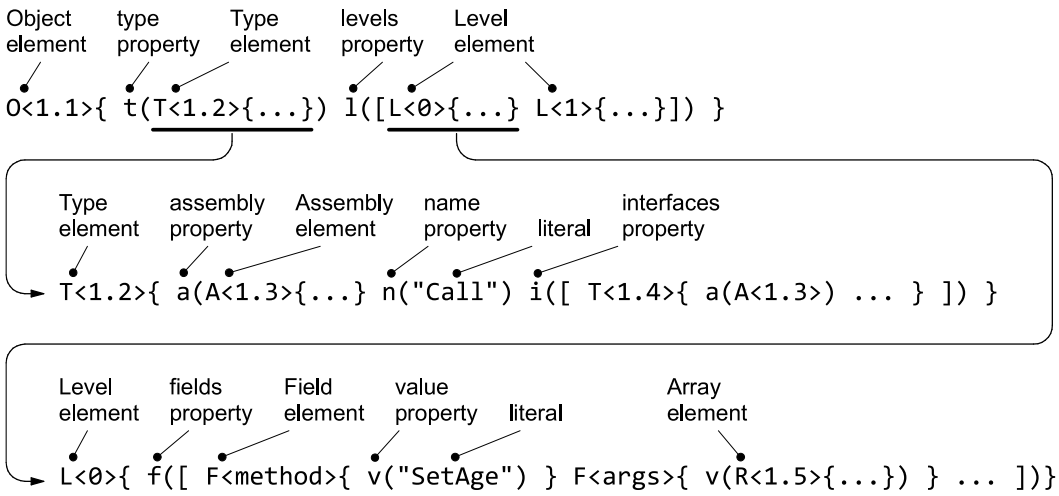


Figure 5.12: Output of the object formatter and the type formatter

The *Level* elements of the *Object* element's *levels* property describe the field values at every inheritance level. The inheritance level is given by the *Level* element's identifier. Level *0* describes the fields of the object's implementation type; level *1* describes the fields of the object's base type. Each *Level* element has a *fields* property that contains a collection of *Field* elements. The identifier of a *Field* element is the field name and a *value* property contains the field value, e.g. level *0* has the *Field* element *method* with a *value* property containing the method name *SetAge* as literal. The *Field* element *args* has a value property with an *Array* element.

5.4.6 Object Reference Identity

The interaction standard specifies object reference identity for distributed objects. For objects that are transferred from one environment to another several times, the remote environment must always get a reference to the same object copy in the case of a serialized object, or the same proxy object in the case of a remote object. Thus, two references that are the same in the original environment must be the same in the remote environment, too. To achieve reference identity, a copy or a proxy is only generated once and is reused for subsequent transmissions. Furthermore, if the object on the remote environment is sent back to the original environment, the original environment must receive the original object, but not a copy or a proxy.

Figure 5.13 on the next page describes how reference identity for distributed objects is implemented by the use of *reference stores*. When an environment sends an object to a remote environment (1), the serializer registers the object in its reference store (2) if it was not already registered. By registering the object it gets a unique object id, which is stored with the object. The identifier consists of the environment id and a consecutive number. In Figure 5.13, the object *A* gets the object id *1.1*. Next, the coordinator transfers the serialized object and its object id to the remote environment (3). When the remote environment receives the object (4), the serializer checks whether the object

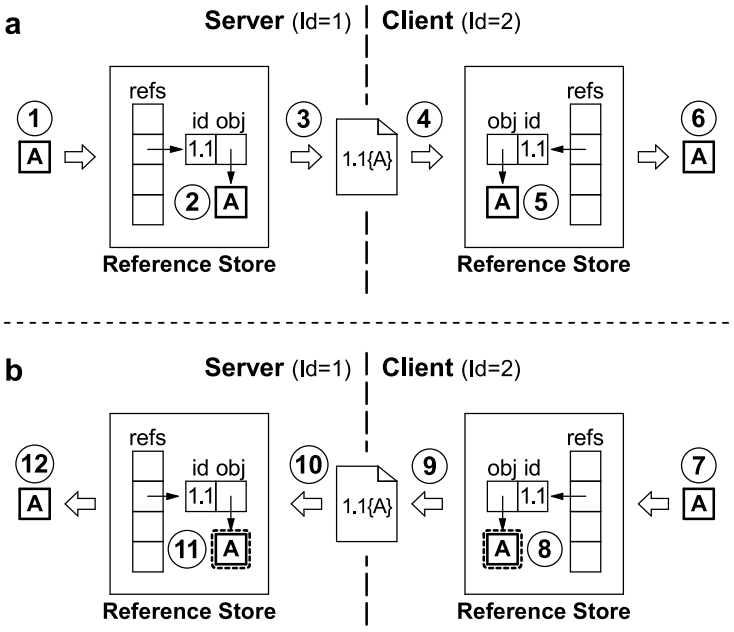


Figure 5.13: Implementing object reference identity by using reference stores

with the id *1.1* is already registered in the remote reference store. If not, the serializer generates a new instance of the object and registers it with its id (5); otherwise it reuses the already registered object. Finally, the remote environment can use the distributed object (6).

When the distributed object is sent back to the original environment (7), the serializer on the remote environment finds the object in the reference store (8) and thus serializes it with id *1.1* and transfers it back (9). The original environment receives the message with the serialized object (10) and finds the object id *1.1* in its reference store (11). Thus, the serializer reuses the original object (12).

5.4.7 Object Data Synchronization

For serialized objects the serializer generates a copy of the original object on the remote environment. If the object gets modified on the remote environment, the original object is out of date. In order to keep the original object synchronized with all copies, Plux updates all modified objects on each token pass. Thus, every object that is accessed within the runtime thread is up to date.

Figure 5.14 shows how object data synchronization is implemented by the use of *profile stores*. When a serialized object is sent to a remote environment (1) the serializer generates a profile for the object (2), which contains a copy of the object's data and a link to the reference in the reference store, which is generated on the first transfer, too (3). Now the object gets serialized (4), transferred to the remote environment, and deserialized there (5). On deserializing the object, the serializer also stores a profile (6)

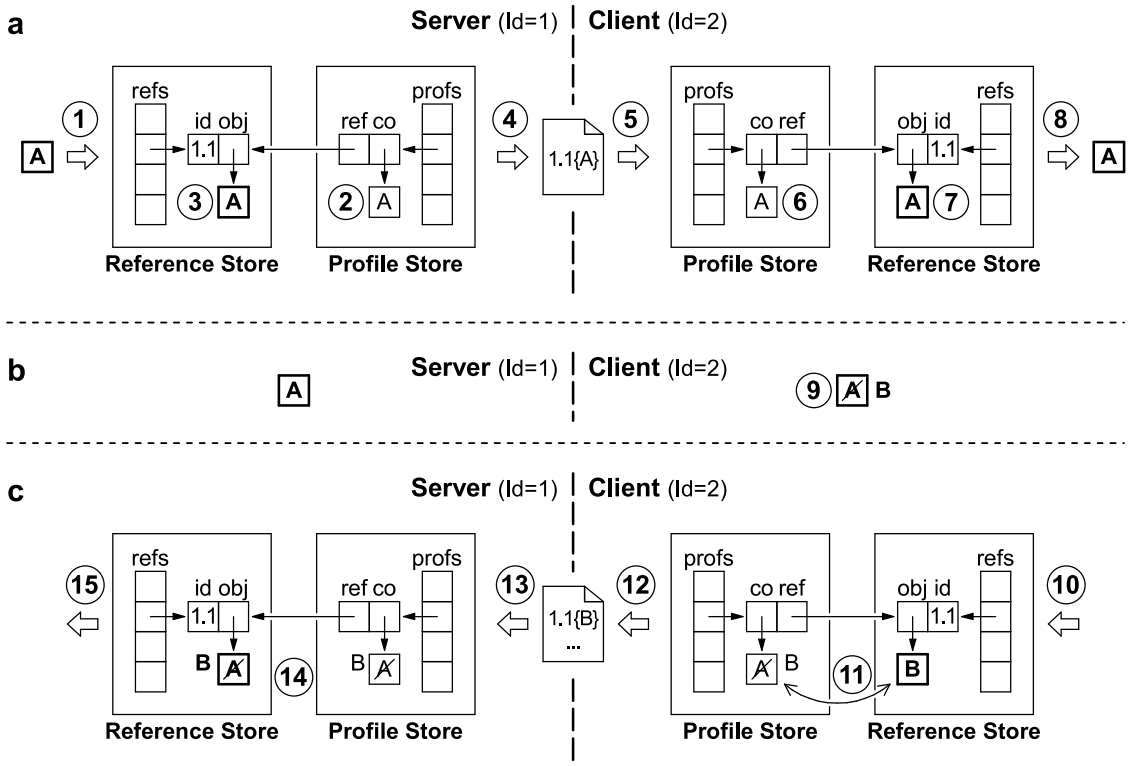


Figure 5.14: Achieving object data synchronization by the use of a profile store

in the remote profile store and registers the object in the remote reference store (7) before the received object can be used on the remote environment (8).

Let's assume that during execution of the received message on the remote environment the data of the serialized object changes from *A* to *B* (9). When a reply message is sent back (10), the serializer compares the values of all objects in the profile store with the values of the corresponding objects in the reference store. If any profile value differs from its object's value, the serializer updates the profile (11) and includes the new values in the reply message (12). When the original environment receives the message (13), it updates all modified objects as well as their profiles (14), before it continues executing in the runtime thread (15).

Incremental Data Transmission

In order to be able to synchronize modifications on serialized objects, every environment stores a profile for the object. The profile always stores the state of the object as it currently is on the remote environment. Thus, when a message is sent to a remote environment, the serializer can check if any object has been modified. Furthermore, if an object is transferred a second time, only the object id and any modifications need to be transferred, but not the whole object. This reduces the communication overhead for serialized objects.

Figure 5.15 shows an example. When the *Server* sends an object with the value *B* to the *Client* (1), it finds the object in the profile store (2). As the object was not modified on the *Server*, the serializer only transfers the object id in the sent message. When the *Client* receives the message (4) it again finds the object in its reference store (5) and can use it without updating the serialized object (6).

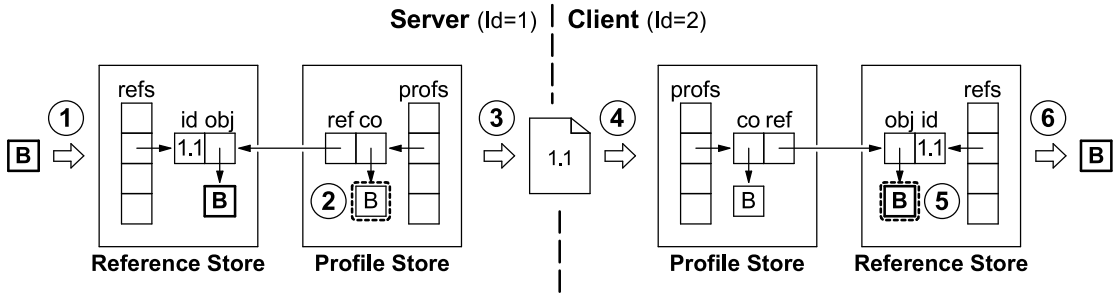


Figure 5.15: Incremential data transmission by the use of profile stores

Multiple Profile Stores

Profiles store the state of a distributed object as it currently is on the remote environment. If an environment is connected to multiple environments, each connected environment can have a different state of a distributed object. Thus, environments need to maintain an individual profile store per connected environment.

In Figure 5.16 the *Web Server* is connected to a *Client* environment and an *Application Server* environment. At the beginning a serialized object with the id *1.1* already was sent to each environment while it had the value *A*. Thus, each profile store has a profile with the value *A* for this object. In (1) the *Client* has the token and modifies the value of

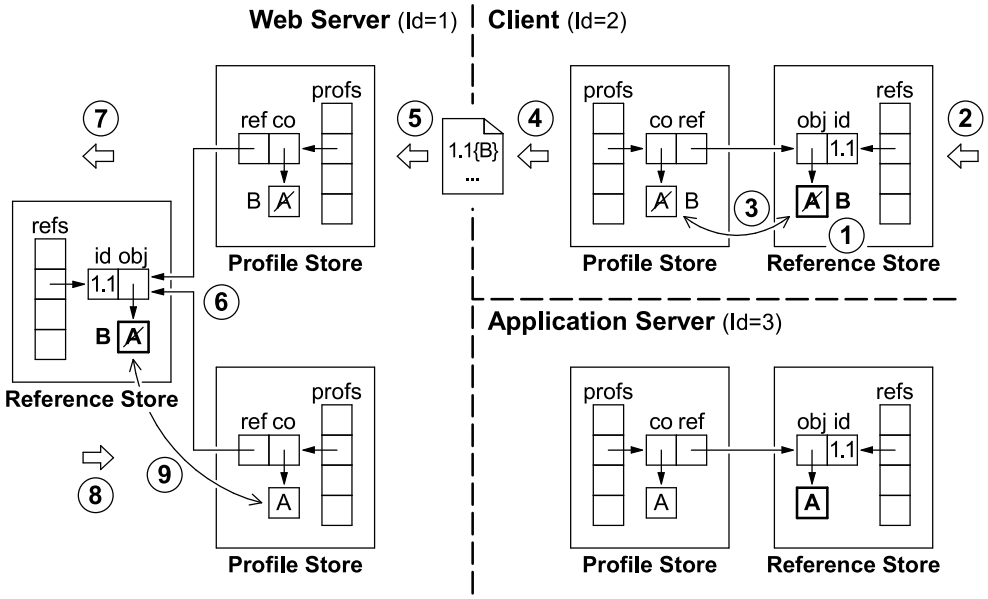


Figure 5.16: Maintaining an individual profile store per connected environment

the object to *B*. Then the token is passed to the *Web Server* (2), whereat the *Client* serializer compares its profiles with the objects in the reference store, updates the modified profiles (3), and writes the modifications to the sent message (4).

When the *Web Server* receives the message (5) it updates the object and also its profile in the profile store for the *Client* (6) before the *Web Server* continues executing the runtime thread (7). In (8) the token is sent to the *Application Server*. Now the serializer compares the objects in the profile store for the *Application Server* with the objects in the reference store (9). As the *Web Server* maintains a special profile store for the *Application Server*, it detects that the serialized object on the *Application Server* is not up to date. Thus, it serializes the modification into the message (not shown in Figure 5.16 as these steps are the same as in Figure 5.14).

5.4.8 Object Lifetime Management

Every distributed object lives on multiple environments at the same time, i.e., the remote environment either has a copy of the original object in the case of a serialized object, or a proxy object in the case of a remote object. Object copies and proxy objects are linked to the original object via reference identifiers, which are maintained by the reference store (see Section 5.4.6 Object Reference Identity). Furthermore, the data of serialized objects is synchronized by the use of profiles, which are maintained by the profile store (see Section 5.4.7 Object Data Synchronization). A distributed garbage collection mechanism manages the lifetime of distributed objects and removes entries in the reference store and profiles in the profile store if they are not required anymore. As distributed garbage collection works differently for serialized objects and for remote objects, its description is separated in two subsections for serialized objects and for remote objects.

Garbage Collection for Serialized Objects

For a serialized object, the entries in the reference store and in the profile store need to be kept on an environment, as long as the object lives on this environment. If a serialized object dies on an environment, the environment can remove its entries in the reference store and in the profile store. As soon as the object lives only on one environment, it is not distributed anymore. Thus, this environment can remove the entries in the reference store and in the profile store for this object, too. In order to let the environment know whether an object is distributed or not, the reference store counts the number remote environments in which an object is currently living.

Entries in the reference store have a reference to a distributed object. In order to allow an object to be destroyed on an environment, the strong reference needs to be changed into a weak reference when the token is passed to another environment. When an environment receives the token, weak references are changed into strong references again. As an environment without the token is considered idle, the local garbage collector is started after each token pass, by which unreferenced objects get collected.

Figure 5.17 shows an example for the distributed garbage collection mechanism. In the example a serialized object is sent from the *Server* environment to the *Client* environment. Before the serialized object is sent to the *Client* (1), the *Server* creates a profile in the profile store (2) and an entry in the reference store (3). As the object is currently living only on the *Server*, the environment counter is initially set to *1*. Similar to the data of serialized objects, this counter also needs to be synchronized with remote

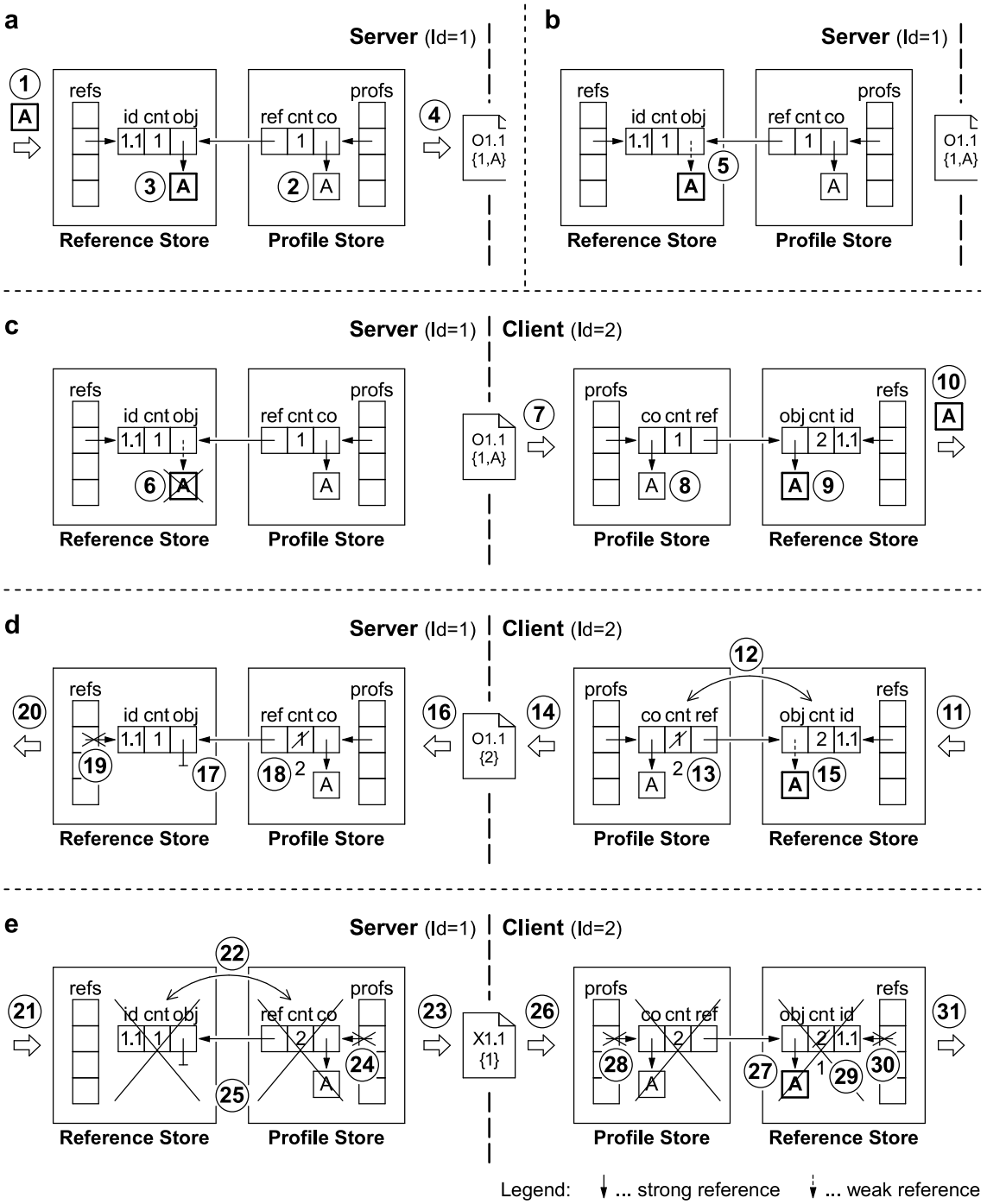


Figure 5.17: Distributed garbage collection for serialized objects

environments. Thus, the counter is stored in the profile for this object, too. Next, the serializer writes the counter and the data of the object to the message to be sent (4). After the message was sent, all references to serialized objects in the reference store are changed into weak references (5) and the local garbage collector gets started. If a serialized object is not referenced by any object on the *Server* anymore, the serialized object gets destroyed (6).

In the meantime the sent message is received on the *Client* (7). The serializer on the *Client* creates an entry for the received object in the profile store (8) and in the reference store (9). As the object was not already registered at the reference store, the counter for the number of remote environments is incremented to *2* in the entry of the reference store. The counter in the profile holds the value that is currently known on the remote environment. Now the object can be used on the *Client* (10).

When the token is passed back to the *Server* (11), the serializer on the *Client* compares all profiles with the entries in the reference store to detect their modifications (12). As the serialized object was not modified but only the counter for remote environments was incremented, the serializer increments also the counter in the profile store (13) and writes the new number of remote environments to the message to be sent (14). After the message was sent, all references to serialized object are changed to weak references on the *Client* (15).

On receiving the message on the *Server* (16), the *Server* changes all weak references to serialized objects back to strong references (17) so that no serialized object can be garbage collected while the environment has the token. This is necessary because if a distributed object would be modified and afterwards garbage collected before the token is sent to another environment, the modification could not be detected on the next token pass. Next, the environment counter of the serialized object gets updated in the profile store to *2* (18). When the serializer updates the environment counter in the reference store, it detects that the object was already destroyed on the *Server*. Thus, it decrements the environment counter to *1*, i.e., it remains the same value as it was before. Furthermore, since the object does not exist anymore on the *Server*, the serializer removes the entry from the reference store (19). However, the profile for the serialized object still has a reference to the entry, which is required when the token is passed to the Client the next time. In (20) execution continues on the *Server*.

When the token is passed to the *Client* in (21), the serializer compares all entries in the profile store with the entries in the reference store (22) and detects that the object was already destroyed. Thus, it writes this information and the updated environment counter to the message (23) and removes the entry for this object in the profile store too (24). This causes that the profile store entry and the reference store entry are destroyed (25). After sending the message, the *Server* again changes all strong references in the reference store to weak references (not shown).

Finally, the *Client* receives the message (26) with the token and changes its weak references in the reference store into strong references (27). On deserializing the

message, the serializer detects the destroyed object and removes its profile form the profile store (28), which causes the entry to be destroyed. Next, the serializer updates the environment counter for the object in the reference store (29). As the number of remote environments is *1* now, the object is not distributed anymore and only lives on the *Client* environment. Thus, the serializer also removes the entry for the object in the reference store (30) and continues execution in the runtime thread (31). As the object is no longer referenced by the reference store, the local garbage collector can destroy the object as soon as it is not used on the *Client* anymore.

Garbage Collections for Remote Objects

The distributed garbage collection mechanism for remote objects differs from the distributed garbage collection mechanism for serialized objects. Remote objects are not copied to other environments, but instead, proxy objects are generated on the remote environments. In order to be able to forward method calls to the original object, Plux must ensure that the original object is not destroyed until all proxy objects are destroyed. The original object as well as all proxy objects are registered in the reference store in order to link them together. However, in order to ensure that original objects are not destroyed prematurely, the reference to the original object in the reference store is never changed into a weak reference. Conversely, as proxy objects cannot be modified and need not be synchronized with the original object, references to proxy objects in the reference store are always weak references. Therefore proxy objects can also be destroyed on an environment that has the token. The original object can be destroyed as soon as all proxy objects have been destroyed. Thus, the entries in the reference store for remote objects have a proxy counter. When the proxy counter is set to *0*, the object is removed from the reference store and can be collected by the local garbage collector.

Figure 5.18 shows an example for the distributed garbage collection of a remote object that lives on the *Server* environment and is accessed by a proxy object on the *Client* environment. When the token is passed from the *Server* to the *Client* (1), the *Server* keeps a strong reference to the original object in the reference store and sets the initial value *0* to the proxy counter (2). The profile store on the *Server* also has an entry for the remote object (3). However, as proxies cannot be modified on a remote environment, the profile only stores the object's proxy counter, as it is known on the remote environment, but not the object's values. On transferring a remote object, the serializer compares its proxy counter in the profile store with the proxy counter in the reference store (4) and sends the counter only if it was incremented or decremented (5).

When the *Client* receives the message with the token (6), it creates a proxy object, adds an entry in the profile store as well as an entry with a weak reference to the proxy in the reference store, and increments the proxy counter to *1* (7). Now the *Client* continues executing in the runtime thread (8). In (9) the proxy object will be destroyed immediately by the local garbage collector as soon as it is not used anymore.

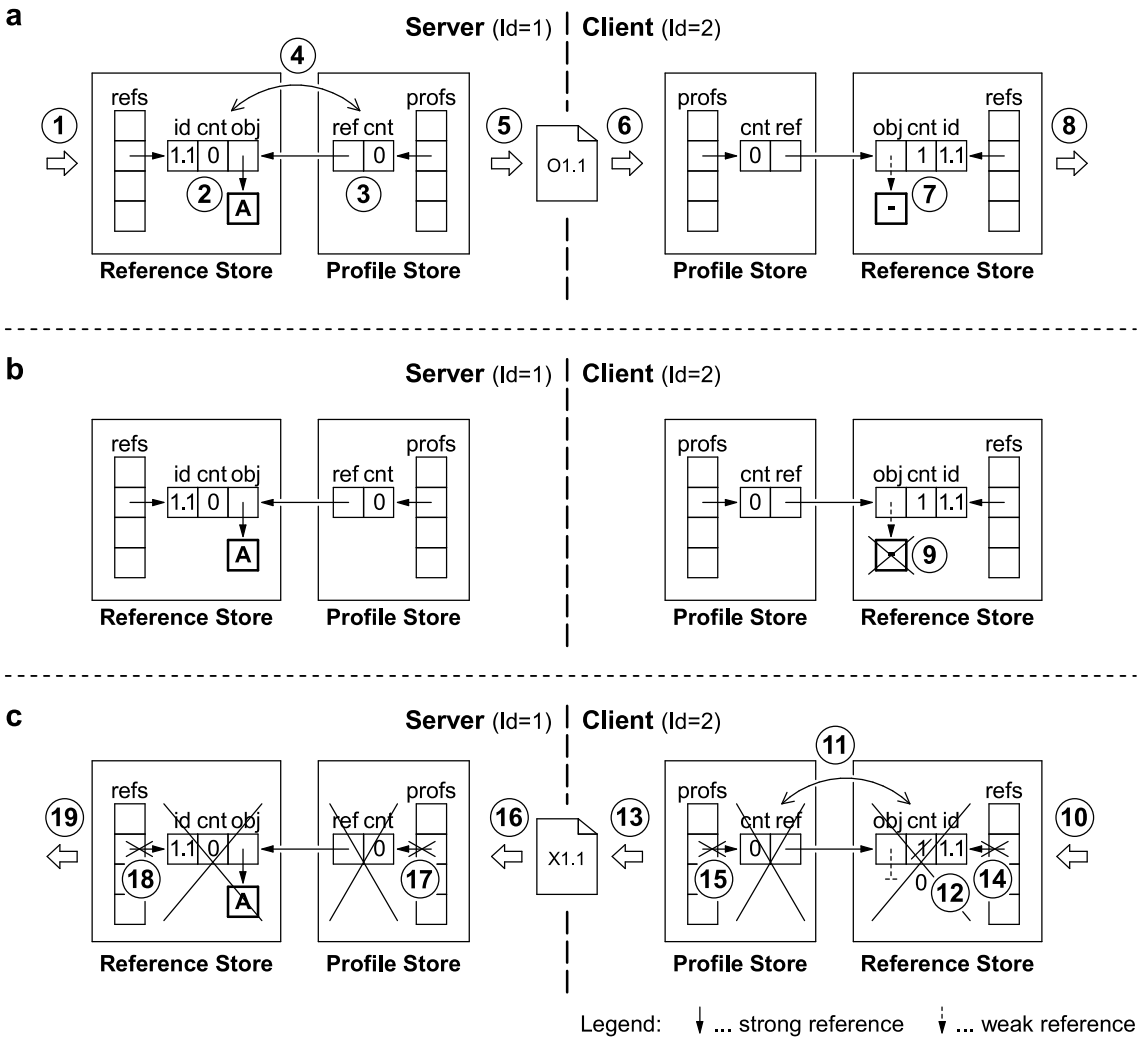


Figure 5.18: Distributed garbage collection for remote objects

When the token is sent back to the *Server* (10), the *Client* detects that the proxy object has been destroyed (11). Thus, it decrements the proxy counter for this object in the reference store (12) and notifies the *Server* that the object with the id 1.1 was destroyed on the *Client* (13). Since the proxy counter in the reference store now is equal to the proxy counter in the profile store, there is no need to send it with the message. Finally, the *Client* removes the entries for the destroyed object in the reference store (14) and in the profile store (15).

When the *Server*, which holds the original object, receives the message with the destroyed object (16), it removes the profile for this object from the profile store (17). Furthermore, as the proxy counter still is set to 0, the *Server* and removes the entry in the reference store (18) too, before it continues executing the received message (19). Now the original object can be destroyed as soon as it is no longer used on the *Server*.

5.4.9 Interoperability

In order to avoid that Plux is limited to a certain technology, the specifications in the interaction standard are not based on technology-dependent communication standards, such as .NET Remoting or .NET binary serialization. Plux uses a text-based transmission format for distributed communication (see Section 5.4.5 Object Transmission Format). Custom formatters (see Section 5.4.4 Object Transmission Mode) enable the interoperability between different technologies. For example, with custom formatters, .NET type information or .NET collection objects can be translated into adequate Java type information or Java collection objects. However, the object data synchronization specification (see Section 5.4.7) requires support for reflection, and the object lifetime management specification (see Section 5.4.8) requires a technology with local garbage collection support.

5.5 Customization Standard

The customization standard specifies a settings model for extensions allowing users and administrators to configure their extensions (see Section 3.5 Customization Standard). As this settings model is the same for web applications, the extended component model for the web does not include any further specifications for the customization standard. However, the current implementation of the customization standard requires that the settings for extensions must be deployed to the same environment as the extensions.

Component Model Implementation

This chapter describes the Plux composition infrastructure, which implements the Plux component model. The composition infrastructure is assembled from several runtime modules, which are exchangeable and can be extended with optional runtime add-ons. The Plux composition infrastructure for the server is hosted and accessed with ASP.NET web pages, the composition infrastructure for the client is provided by browser plugins for different browsers, by a client-side standalone implementation, and by a Silverlight implementation, which is deployed on the server.

The Plux composition infrastructure, which implements the specifications of the Plux component model, allows executing distributed multi-user web applications built from Plux components. The following sections describe the ingredients of the composition infrastructure and its architecture.

6.1 Composition Infrastructure

The composition infrastructure of Plux, which implements the Plux component model, consists of the *Server Runtime* and several *Client Runtimes*, each having its own *Discoverer* implementation (see Figure 6.1 on the next page). The *Server Runtime* enables the execution of component-based web applications that are built from Plux extensions. It hosts an individual *Runtime Node* per user, which assembles and maintains the composition for one user. Furthermore, each *Runtime Node* can be connected to remote *Runtime Nodes*. Every connected set of *Runtime Nodes* constitutes a coherent composition infrastructure, which assembles and maintains a single composition state for a distributed web application. Remote *Runtime Nodes* are hosted by *Client Runtimes*, which can be implemented as web browser plugins, as *Silverlight* applications, or as standalone applications. A *Runtime Node* is connected to the *Server Runtime* when the web browser requests a Plux web application, when a sandbox plugin is composed for the first time, or when the standalone *Client Runtime* is started (see Section 5.4.2 Connection Establishment on page 117). The Server Runtime's *Connection Listener* allows Client Runtimes to connect their Runtime Node to the Runtime Node on the server. Client Runtimes may also have a Connection Listener in

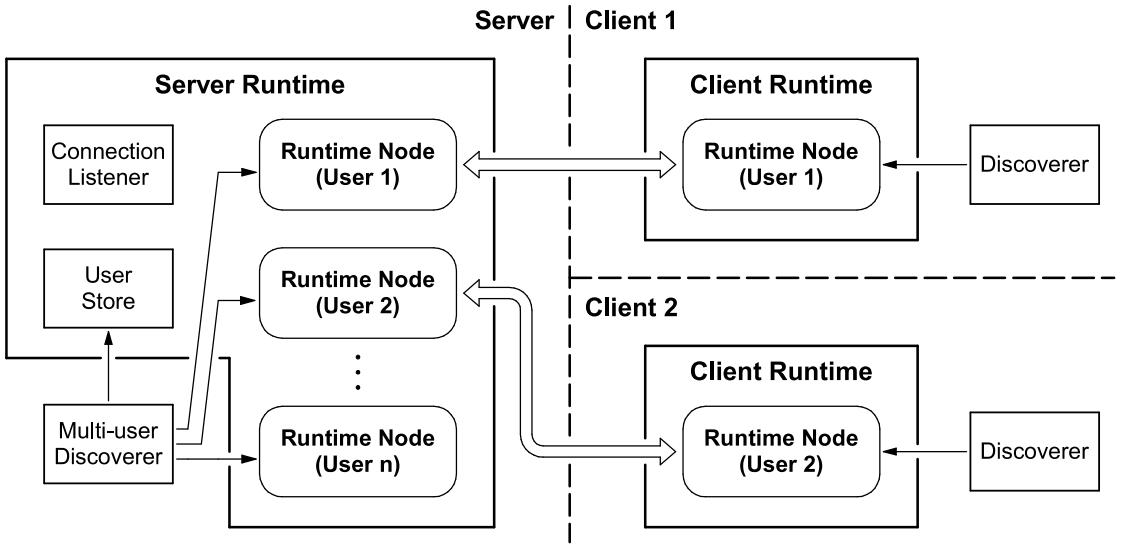


Figure 6.1: Plux composition infrastructure

order to connect themselves to each other. The *User Store* maintains the hierarchy of users and user groups. It is used by the *Multi-user Discoverer*, which assigns server-side plugins and sandbox plugins to the respective *Runtime Nodes*. The client-side *Discoverer* discovers client-side plugins and adds them to the composition infrastructure via the *Runtime Node* of the *Client Runtime*. The server-side discoverer and the client-side discoverers are not part of the Plux runtime, but are implemented as exchangeable extensions (see Section 3.2 Deployment Standard on page 43).

Runtime Architecture

Every runtime node consists of several runtime modules, each of which implementing a specific part of the component model. As all runtime modules are exchangeable, the component model implementation is adaptable and can be extended with optional runtime modules (see Section 6.2 below). How runtime modules can be exchanged is described in Appendix A.2: Runtime Configuration. Figure 6.2 shows the runtime modules that constitute a runtime node. The core modules implement the base composition model for component-based desktop applications and can be used independently from the web modules. The web modules implement the extended component model for the web and enable support for component distribution.

The external *Discoverer* discovers new plugins and contracts in a background thread and uses the *Dispatcher* (1) to add them to the *Type Store* (2) in the runtime thread. The *Type Store* allows storing and retrieving type information for extensions and triggers the *Composer* to compose new extensions (3) when new plugins were added. The *Composer* queries the current composition state in the *Instance Store* (4) to find matching slots for the plugs of new extensions. The *Instance Store* maintains the composition state of an application and stores all instances of extensions and their connections. If the *Composer* finds a matching slot for a new extension, it instructs the *Qualifier* to check

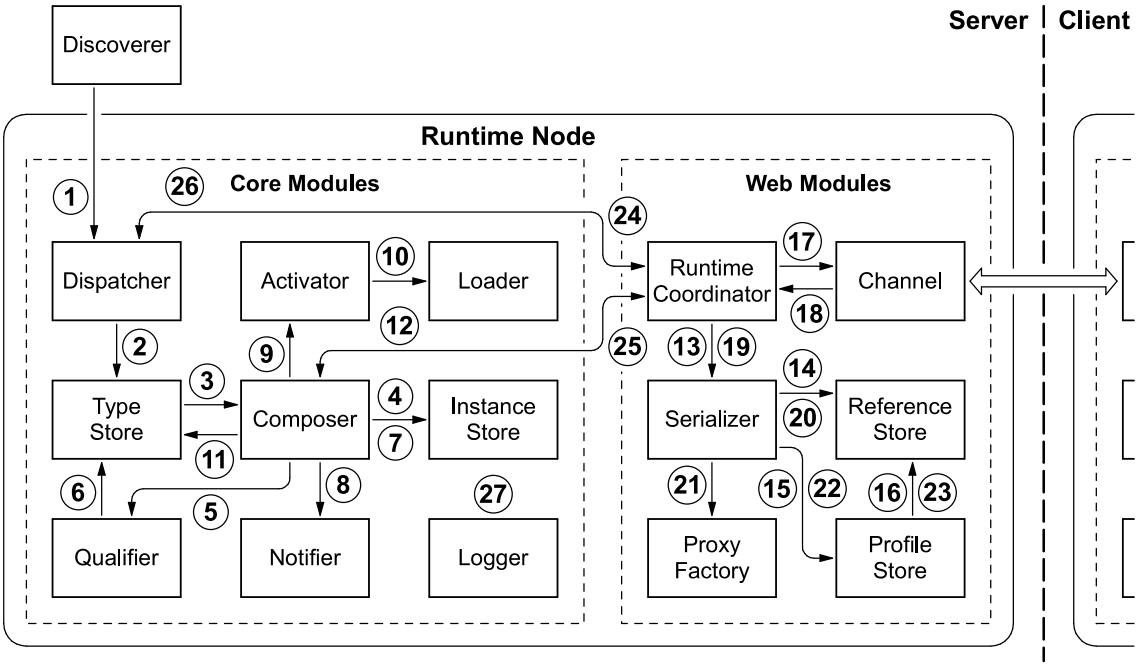


Figure 6.2: Plux runtime modules

whether the extension with the matching plug implements the required interfaces and whether it provides the required parameters (5). For this, the *Qualifier* retrieves the slot definition for the matching slot from the *Type Store* (6) and compares the slot definition's requirements with the extension's provisions. If the extension is qualified to be composed, the *Composer* plugs the matching plug into the slot and updates the composition state in the *Instance Store* (7). The *Composer* uses the *Notifier* (8) to raise appropriate composition events during composition operations. The *Notifier* maintains all registered event handlers, regardless of if they were registered by the extensions' meta-objects or by composition behaviors. If some code accesses the extension object of an extension, e.g., in an event handler of a *Plugged* event, the *Composer* starts a new composition sequence for the extension and instantiates its extension object by the use of the *Activator* (9). The *Activator* uses the *Loader* to load the assembly that implements the extension (10), if it was not already loaded. The *Loader* retrieves the location of the assembly via an assembly Uri that is stored in the extension's meta-object, and possibly downloads the assembly from a different computer before it is loaded. For example, the *Loader* of the *Silverlight* runtime needs to download assemblies from the server before it can load them on the client-side *Silverlight* environment. After activating the extension, the *Composer* opens the extension's slots and again uses the *Type Store* (11) to find contributors for the extension's slots. It composes them by repeating steps (4) to (11) recursively, until all slots have been filled with matching extensions.

If the *Composer* needs to start a composition sequence for a remote extension, it forwards the composition operation to the remote environment via the *Runtime Coordinator* (12). The *Runtime Coordinator* serializes the message with the *Serializer* (13).

The *Serializer* uses the *Reference Store* to register distributed objects and to get a unique object id for them (14). Furthermore, the *Serializer* uses the *Profile Store* to create and update profiles for serialized objects (15) in order to implement object data synchronization. The *Profile Store* uses the *Reference Store* to link profiles to their original object (16). Finally, the *Runtime Coordinator* uses a *Channel* (17) to send a message to the remote environment.

When the *Runtime Coordinator* receives a message via a *Channel* (18), it uses the *Serializer* (19) to deserialize the message. The *Serializer* uses the *Reference Store* (20) to query received object ids and to reuse existing objects that were already registered. If an object was not yet registered in the *Reference Store*, the *Serializer* creates a new instance of the distributed object and registers it in the *Reference Store*. For instantiating proxy objects, the *Serializer* uses the *ProxyFactory* (21), which dynamically generates proxy types from interface descriptions and uses them to instantiate proxy objects. On deserializing a message, the *Serializer* uses the *Profile Store* (22) to update profiles and objects that were modified on the remote environment, or to create new profiles if they do not already exist. The *Profile Store* links profiles to their object in the *Reference Store* (23). As soon as the message is deserialized, the *Runtime Coordinator* uses the *Dispatcher* (24), if the message needs to be executed in the runtime thread. If the received message is a composition operation to be performed, the *Runtime Coordinator* forwards the operation to the *Composer* (25). Finally, the *Dispatcher* forwards dispatcher operations to remote environments using the *Runtime Coordinator* (26), if they need to be enqueued by the *Dispatcher* there.

All runtime modules and all extensions can use the *Logger* (27), e.g., to log an upcoming task or to log that a task is finished. The logger interface provides different verbosity levels for log messages: *Quiet*, *Minimal*, *Normal*, *Detailed*, and *Diagnostic*. The sender of a log message can choose the verbosity level for each log message. Whether the message gets logged depends on the verbosity level that is set at the runtime configuration. Extensions access the *Logger* via their meta-objects.

6.2 Runtime Add-ons

Runtime add-ons are optional runtime modules that can be installed to extend the Plux runtime. The mandatory runtime modules provide hooks to which runtime add-ons can listen and by which they can influence their behavior. This section describes the optional runtime modules *Security Add-on*, *Testing Add-on*, and *Debugging Add-on*, which are provided by the composition infrastructure.

Security Add-on

By default, the Plux composition infrastructure allows adding any plugin to the type store. When composing an application, it connects all matching slots and plugs, and it gives all extensions full permissions. The *Security Add-on* can be installed on the runtime in order to refuse unauthorized plugins, to block illegal composition

operations, to isolate untrusted plugins in sandboxes, and to wire interceptors between extensions. In order to do so, the *Type Store* asks the *Security Add-on* if it is allowed to add a plugin; the *Composer* asks the *Security Add-on* if it is allowed to connect two extensions; if the connection is allowed, the *Security Add-on* may nevertheless wire an interceptor between the extensions in order to restrict the interface of an extension; and the *Security Add-on* may restrict the permission set of an extension when it is created (e.g., by denying access to the hard disc or to the network). Finally, the *Security Add-on* can also enforce restrictions on disconnecting and removing extensions.

The *Security Add-on* obtains security settings either from a configuration file or from attributes that can be attached to extensions, slots and plug, which allows manufacturers to specify restrictions directly in the source code of the extensions. Custom security scenarios, which are not covered by the configuration file or by attributes, can be covered by implementing security libraries. The *Security Add-on* is described in detail in [Wolfinger et al., 2012].

Testing Add-on

Plux allows dynamic reconfiguration of applications by dynamically adding or removing components. To test whether a component is dynamically composable, it is not sufficient to test it in isolation, but one also has to test it in combination with other components and with dynamic reconfiguration. The *Testing Add-on* allows systematic composability tests by permutating all possible ways in which components can be connected. It reads test specifications from a configuration file, prepares factories to create testbed components, adds testbed components to the type store, uses the composer to execute composition operations, and executes tests on different composition states. [Löberbauer, 2012; Löberbauer et al., 2010]

Debugging Add-on

The *Debugging Add-on* records the composition process of a program, analyzes the composition sequences and composition states, and hints at possible causes of composition errors (e.g., when necessary extensions are missing or when extensions where plugged in the wrong order). Composition errors can be located with a post mortem composition debugger, i.e., by replaying the composition operations and searching for the causes of composition errors.

An additional debugging tool allows developers to evaluate the composition trace, to filter composition operations, to split composition traces into parts that contain related composition operations, to compare composition traces and to visualize differences between them. This tool generates hints for possible error causes using reasoning. [Löberbauer, 2012; Lengauer, 2012]

6.3 Runtime Libraries

Runtime libraries support developers with prefabricated implementations for common tasks in developing component-based web applications. This section describes the *Administration Library*, which provides support for user management and plugin deployment, the *Web UI Library*, which enables developers to build user interfaces from distributed components, and the *Layout Library*, which supports the developer by arranging controls of component-based user interfaces.

Administration Library

The *Administration Library* consists of a set of assemblies that support the administration of user accounts and the deployment of user-specific plugins. It contains different user store implementations that can be used by the server runtime: the XML user store maintains user information in an XML file; the database user store can be used to store user information in a database; and the ASP.NET user store implements the ASP.NET membership API, which can be used to store users in files or in a database, and which can be used for authentication in ASP.NET web applications. Furthermore, the *Administration Library* provides support for deploying plugins to the server-side plugin repository. It allows installing server-side plugins by administrators, who have access to the server environment, and by users, who can install server-side plugins from remote environments.

Web UI Library

The *Web UI Library* allows developers to build distributed user interfaces for web applications from web controls such as *Buttons*, *Labels*, or *Panels*. The user interface for Plux web applications can also be built from ASP.NET web controls, however ASP.NET web controls require to be executed in the AppDomain in which an ASP.NET web application was started. Therefore, with ASP.NET controls it is neither possible to separate user-specific plugins that implement parts of the user interface in different AppDomains, nor is it possible to extend the user interface with distributed components. Therefore, the *Web UI Library* provides an implementation of web controls, which are similar to the ASP.NET web controls, but can be executed in different AppDomains and can be used in the same way in local components as well as in components that are executed on a remote environment.

Layout Library

In component-based web applications, which have the goal to be adaptable and extensible, the user interface should be adaptable and extensible, too. Therefore, the user interface also needs to be built from separate components. However, building user interfaces from separate components, which do not know each other, leads to additional problems that need to be solved. In common user interfaces, developers arrange user controls either by specifying absolute values for the size and position of a control, or by specifying those values relative to other controls, which are known by

the developer. As user controls in component-based user interfaces do not know the other controls in the current composition, it is not possible to arrange controls with static values for their size and position. The *Layout Library* implements a solution for dynamically arranging user controls in component-based user interfaces. For this, UI components specify layout information declaratively in their metadata via .NET attributes. The layout information comprises generic specifications for the size and the location of every control. This is evaluated by the *Layout Library* to arrange all existing controls in a coherent user interface.

Chapter 7

Evaluation

This chapter evaluates our approach of building plugin-based distributed multi-user web applications with Plux. For our evaluation, we adapted two existing plugin-based desktop applications so that they can be deployed as web applications. In this process, we analyzed the degree of component prefabrication and component reusability.

The purpose of component-based software development is to build applications from loosely coupled components with well-defined interfaces, which can be implemented independently. The goal of this approach is to prefabricate and to reuse components in different applications. Furthermore, plugin components enable applications to be customizable and extensible with third-party plugins provided by end users.

The following sections evaluate the degree of component prefabrication and component reusability of plugin-based distributed multi-user web applications with regard to the component model for the web, which was presented in this thesis. For this, we analyzed two case study applications, which originally were implemented using the original Plux component model that was introduced in the PhD thesis of Wolfinger [Wolfinger, 2010]. Since this component model targets single user desktop applications only, both case study applications initially were desktop applications.

One of our research goals was to increase component reusability when building web applications from existing desktop applications. Therefore, for evaluation purposes we adapted both case study applications so that they can be used as web applications as well. In doing so, we determined the number of components that could be reused for the web without any modification, the number of generic web components that were already prefabricated, and the number of components that needed to be re-implemented for the web.

A further goal was to increase customizability and extensibility for component-based web applications. By supporting distributed components, we enabled users to customize the architecture of their applications according to their needs. Applications now can be deployed as pure web applications, where the whole application is deployed and executed on a web server; they can be deployed as rich web applications, where all components are installed on a web server, but the components for the user

interface are transferred to the client computer and executed there when the application is accessed, while the components for the business logic stay on the web server; they can be deployed as thin client applications, where only the business logic components are installed and executed on a server whereas components for the user interface are installed and executed on the client computer; or they can be deployed as pure desktop applications, where all components are installed and executed on the client computer. Furthermore, the extensibility of web applications was increased, since users now can extend them with their individual user-specific components either on the server-side or on the client-side. However, it is difficult to determine the degree of how much the extensibility of the applications was increased, because existing component models do not support user-specific extensions for web applications at all.

Section 7.1 describes the extensions and the composition architecture of the case study applications that were used for evaluation and Section 7.2 presents the degree of component prefabrication and reusability that could be achieved when adapting both case studies to deploy them as pure web applications, as rich web applications, and as thin client applications.

7.1 Case Studies

For evaluation purposes, we adapted two existing plugin-based desktop applications in order to deploy them as web applications. The first case study is the *Time Recorder* application for recording working hours, which was already introduced in Chapter 4 to motivate the approach for building plugin-based distributed multi-user web applications. The second case study is a *Cross Compiler* with an IDE, which was originally implemented for a Master's Thesis in the context of the Plux research project [Jahn, 2009]. The *Cross Compiler* case study implements a plugin-based compiler that translates source code from one programming language or markup language into another language. The input language as well as the output language can be customized by using different components for parsing the input and generating the output. Both applications consist of multiple extensions, which can be distributed across multiple computers and can be extended by end users.

The following subsections describe the composition architecture of the case study applications. We classify extensions into *generic* extensions and *specific* extensions. *Generic* extensions can be prefabricated and reused in any application, while *specific* extensions are implemented exclusively for a specific application. Additionally, all extensions are assigned to one of the following application layers. The *System layer* consists of extensions that are provided and used by the Plux composition infrastructure, the *Presentation layer* consists of extensions that implement the user interface of the application, *Application layer* extensions implement the business logic of an application, and *Data layer* extensions provide access to the data that is used by an application. Besides the classification into *generic* and *specific* extensions, the application layer of an extension influences its degree of component reusability as well.

7.1.1 Time Recorder

The *Time Recorder* application provides features for recording and evaluating working hours. Figure 7.1 shows its user interface, which is composed from multiple user controls that are implemented as independent extensions. It consists of the menu (1), the recorder for starting and stopping time records (2), the project recorder for assigning time records to different projects (3), the notes control for attaching notes to the current time record (4), the status area for displaying status information for the current time record (5), and the presence view for displaying time records in a selected time range (6).

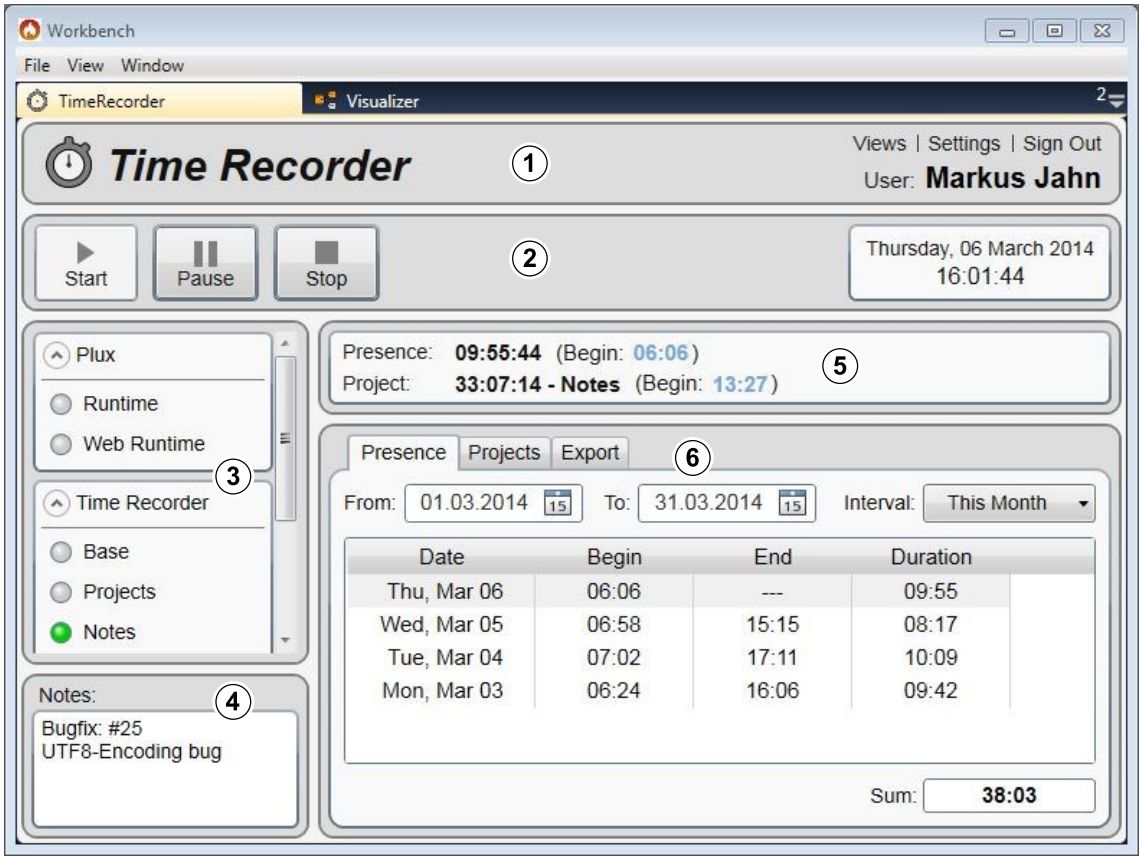


Figure 7.1: User interface of the *Time Recorder* application

Figure 7.2 on the next page shows the composition architecture for the frontend of the *Time Recorder* application. The figure contains the extensions in the *Presentation layer*, which implement the user interface shown in Figure 7.1, and it contains the extensions in the *System layer*, which implement the exchangeable discovery mechanism. All extensions in the *System layer* are *generic*, while the extensions in the *Presentation layer* are separated into *generic* and *specific* extensions. The composition architecture for the backend is shown in Figure 7.3 on page 151. Those extensions are separated into *Application layer* extensions, which implement the business logic of the application, and

into *Data layer* extensions, which provide access to the data that is used by the business logic extensions. Most backend extensions are *specific* extensions, which are implemented exclusively for the *Time Recorder* application, but a single extension in the *Data layer* is generic, which can be reused in any application.

System layer (generic)

The *System* layer consists of the generic extensions *Discoverer*, *Filesystem Detector*, and *Assembly Analyzer*, which implement the exchangeable discovery mechanism. These extensions were described in Section 3.2 and can be reused in any application.

Presentation layer (generic)

All *generic* extensions in the *Presentation layer* are provided by the Plux composition infrastructure. The *Workbench* extension acts as a host for view extensions, which can be opened within the display area of the workbench. For arranging views, the *Workbench* uses a *TabContainer* extension. Furthermore, the *Workbench* provides a slot for a *Menu* extension, which is displayed on the top of the display area, as well as a slot

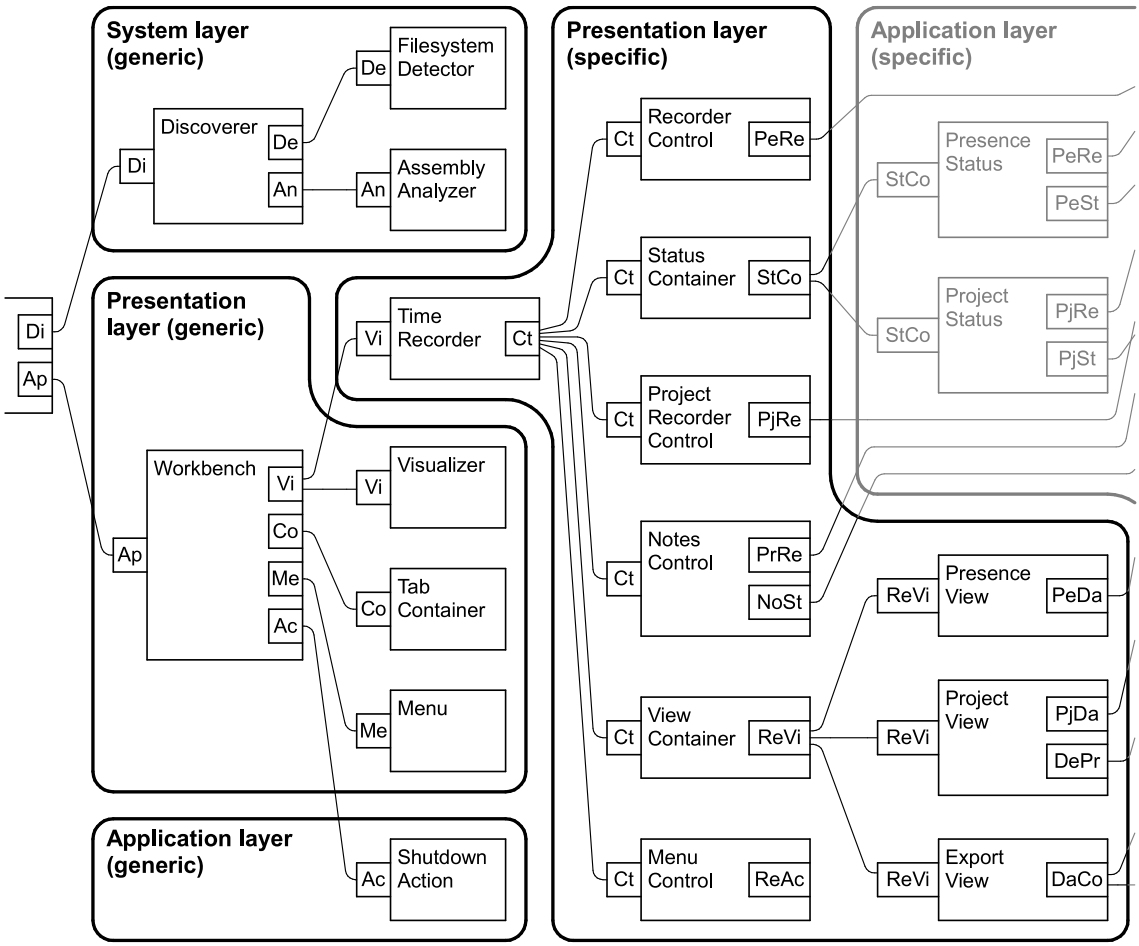


Figure 7.2: Frontend composition of the *Time Recorder* application

for *Action* extensions, which are displayed as entries in the menu. The *Visualizer* is a *View* extension that shows the current composition state of the application and provides an interface for modifying the current composition state (see Section 3.3.7).

Presentation layer (specific)

The *specific* extensions in the *Presentation layer* implement the user interface of the *Time Recorder* view (see Figure 7.1), which is plugged into the *Workbench*. The *Time Recorder* extension is the host for control extensions, which provide the user interface for the various features of the application. The *Recorder Control* contains buttons for starting, pausing, and stopping recording; the *Status Container* provides an area for displaying status information such as duration and start time of the current time record or the sum of recorded working hours for the current project; the *Project Recorder Control* provides a list of radio buttons that assign the current time record to a particular project; the *Notes Control* allows the user to enter a note for the current time record; the *View Container* provides a display area within the *Time Recorder* to show the content of internal views; the *Presence View* shows a list of time records for a certain period of time; the *Project View* shows time record statistics for a selected project; the *Export View*

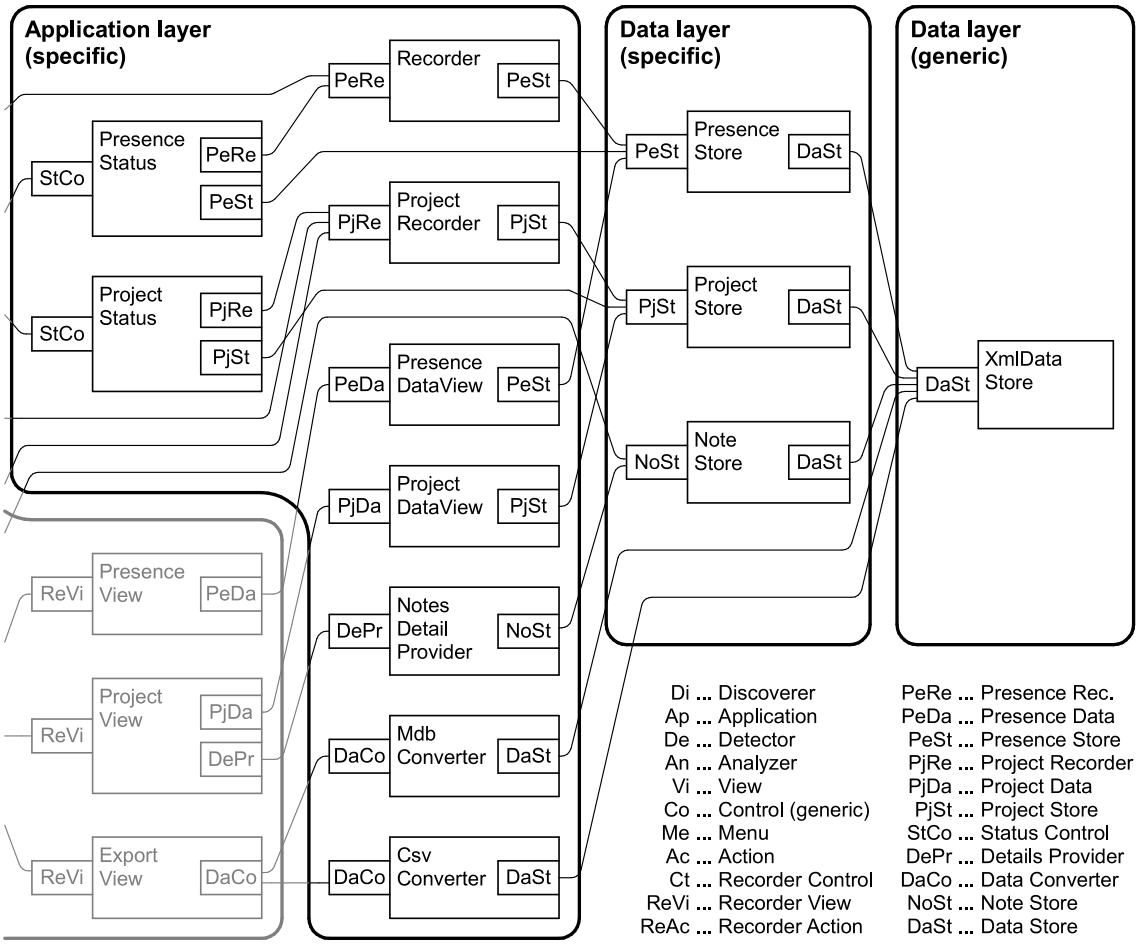


Figure 7.3: Backend composition of *Time Recorder* application

allows users to export time records into a file; and finally the *Menu Control* provides an internal menu that shows entries for actions, which can be plugged in its *TimeRecorderAction* slot.

Application layer (generic)

The *Application layer* consists only of a single *generic* extension, which is the *Shutdown Action*. It is plugged into the *Workbench* and it is used to close the application. Since the implementation of the extension just invokes the runtime's *Shutdown* method, but does not provide a user interface, it belongs to the *Application* layer.

Application Layer (specific)

Specific extensions in the *Application layer* are used by *specific* extensions in the *Presentation layer*. The *Recorder* is used by the *Recorder Control* to start and stop recording working hours; the *Presence Status* is displayed by the *Status Container* and uses the *Recorder* to determine the start time as well as the duration of the current time record and it uses the *Presence Store* of the *Data layer* to sum up all time records of the current day; the *Project Recorder* is used by the *Project Recorder Control* to assign time records to different projects, it is used by the *Notes Control* to assign notes to the current time record for a project, and it is used by the *Project Status* to display status information for the current project; the *Project Status* is displayed by the *Status Container* and uses the *Project Recorder* and the *Project Store* to sum up the time records of the current project; the *Presence Data View* is used by the *Presence View* and allows selecting, filtering, and sorting time records to be displayed; the *Project Data View* is used by the *Project View* and works similar to the *Presence Data View* for project time records; the *Notes Detail Provider* is used by the *Project View* and provides the notes that are assigned to the selected project time record; and finally the *MdbConverter* as well as the *CsvConverter* are used by the *Export View* to convert stored data into either a mdb file or a csv file.

Data layer (specific)

The *Data layer* contains extensions that allow storing and retrieving data that is used by the application. The *Presence Store* provides access to presence time records; the *Project Store* provides access to project time records; and the *Note Store* provides access to notes for project time records. All the *specific* stores use the *generic* store extension *Xml Data Store*, which provides an interface for storing and retrieving arbitrary data.

Data layer (generic)

The *generic* Data layer contains the *Xml Data Store* extension for writing and reading data into and from an XML file. Besides the *Xml Data Store*, there is also an optional *Database Data Store*, which stores the data into a connected database (not shown).

7.1.2 Cross Compiler with IDE

The *Cross Compiler* application can be used to translate any programming or markup language into any other language. It was developed in the context of the Plux research project in order to translate Delphi source code into C# source code. The component-based architecture of the compiler allows exchanging all its essential parts so that the translation of other languages can be supported as well.

Figure 7.4 shows the graphical user interface of the IDE for the cross compiler, which is again implemented with independent extensions. The *Menu* (1) shows menu entries for actions that can be plugged into the extension that implements the menu; the *Tool Bar* (2) allows controlling the compiler, shows the location of the source file and the output file, and allows selecting the parser and the code generator that are used for compilation; the *Source View* (3) shows the source code of the input; the *AST View* (4) shows the abstract syntax tree that was generated during parsing the source code; the *Symbol Table View* (5) shows the contents of the symbol table; the *Output View* (6) shows the generated output; and the *Error View* (7) shows a list of messages, warnings, and errors, which were logged during compilation.

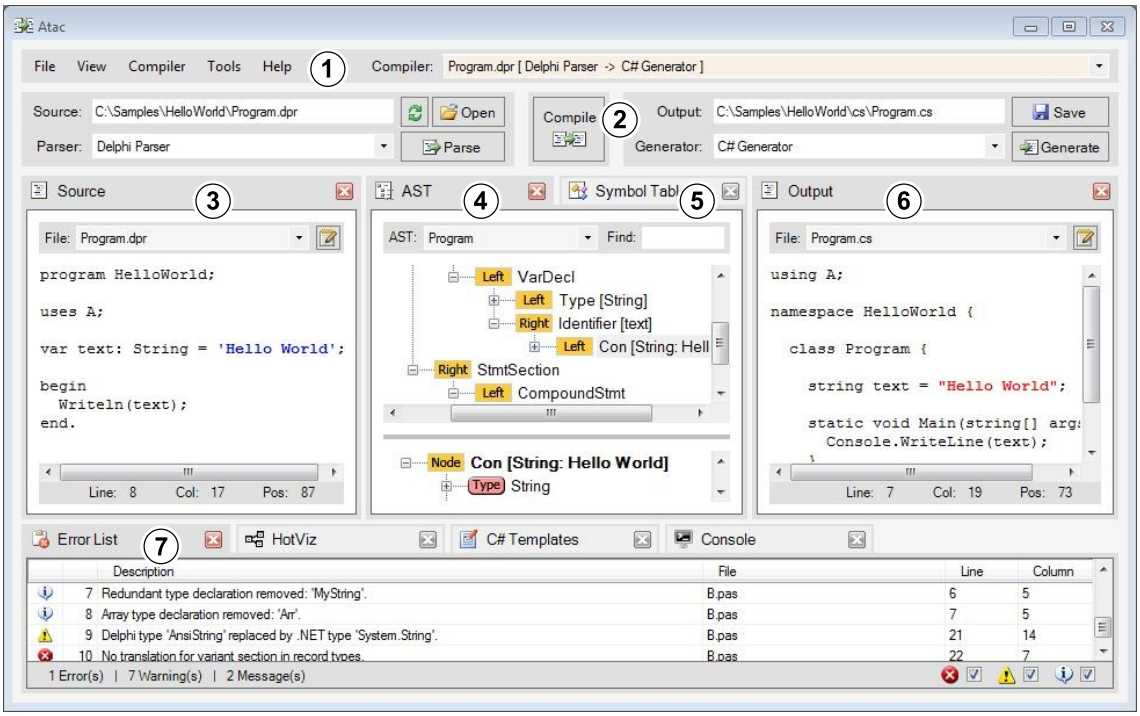


Figure 7.4: User interface of the IDE for the *Cross Compiler* application

The composition architecture of the compiler and its IDE are shown in Figure 7.5 on the next page. Since all *generic* extensions of the *System layer* and the *Presentation layer* of this application are the same as in the *Time Recorder* application, they are not shown in the figure.

Presentation layer (specific)

The cross compiler application provides two different user interfaces. The *Console* extension is used by the command shell and allows users to control the *Compiler* via shell commands. The *IDE* provides a graphical user interface and is implemented as a view extension that can be plugged into the *Workbench* extension (not shown). The *IDE* extension has a slot for the *Compiler* to be controlled. It is extended by several UI extensions, which are displayed within the *IDE* view (see Figure 7.4). The *Menu* shows menu entries, which can be plugged into its *Action* slot. The base deployment configuration of the application contains extensions for the actions *New*, *Open*, *Save*,

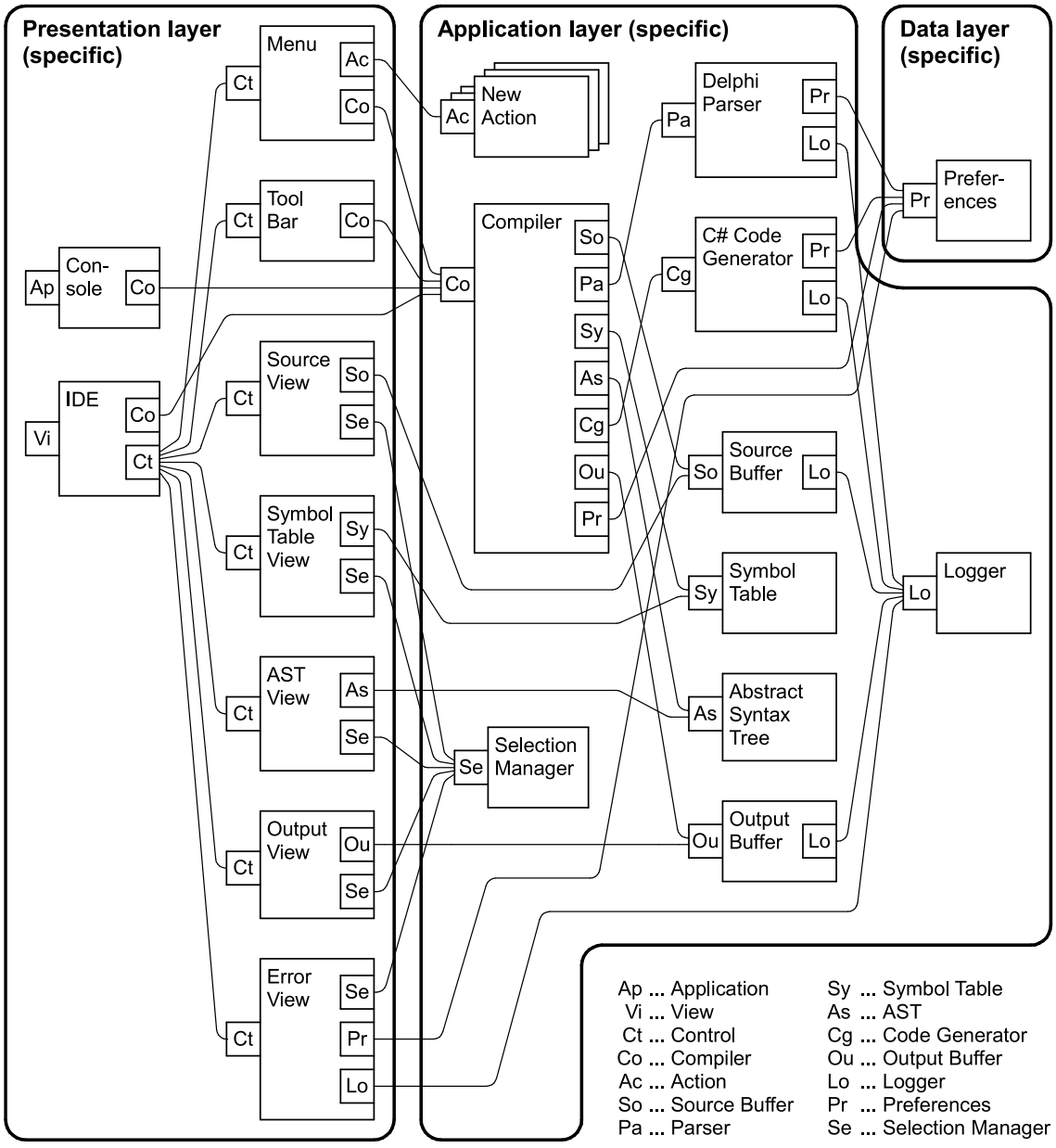


Figure 7.5: Composition of *Cross Compiler* case study

SaveAs, *Exit*, *Parse*, *Compile*, *Generate*, *Reset*, and *About* (not shown). The *Tool Bar* allows selecting the parser extension as well as selecting the generator extension, which are used by the *Compiler* in order to change source and output language. Furthermore, it provides the buttons *Parse*, which starts the parser, *Generate*, which starts the code generator, and *Compile*, which starts the parser and the code generator. The *Source View* displays the contents of the opened source file; the *Symbol Table View* displays the contents of the symbol table, which is filled by the compiler during parsing the source; the *AST View* displays the abstract syntax tree (AST), which is also built up during parsing the source; the *Output View* displays the output that is generated by the compiler; finally, the *Error View* displays messages, warnings and errors that occurred during compilation.

Application layer (specific)

The *Application layer* consists of extensions that implement the functional part of the compiler. The *Compiler* extension combines the ingredients of a compiler that are necessary for translating source code from one language to another. The ingredients are implemented as extensions, which are plugged to the *Compiler*. The *Source Buffer* provides the source code to be translated. The *Delphi Parser* analyses Delphi source code, fills the *Symbol Table*, and builds up the *Abstract Syntax Tree*. The *Symbol Table* stores all identifiers of the source code and their meanings. The *Abstract Syntax Tree* models the source code in a tree-like data structure. The *C# Code Generator* uses the *Abstract Syntax Tree* and the *Symbol Table* to generate C# code, which is written into the *Output Buffer*. The extensions *Compiler*, *Source Buffer*, *Delphi Parser*, *C# Code Generator*, and *Output Buffer* use the *Logger* to log messages, warnings and errors, which are displayed by der *Error View*. Finally the *Selection Manager* is used by the *Source View*, the *Symbol Table View*, the *AST view*, the *Output View* and the *Error View* to synchronize the selection in those views. This allows the user to select source code in the *Source View*, to see the generated output for the selection in the *Output View*, and to see the internal data of the compiler for the selection in the *Symbol Table View* and in the *AST View*.

Data layer (specific)

The *Data layer* of our cross compiler consists only of the extension *Preferences*, which stores compiler configurations such as the location of symbol libraries for the symbol table.

7.2 Component Prefabrication and Reusability

This section evaluates the degree of component prefabrication and reusability that was achieved by adapting our case studies from desktop applications to web applications. Whether an extension can be prefabricated or reused both in desktop and in web applications depends on whether it is *generic* or *specific* and to which application layer

it belongs. Thus, the following subsections determine the degree of component prefabrication and reusability based on the total number of extensions in the different application layers for generic and specific extensions.

Figure 7.6 shows the number of generic and specific extensions in each application layer as absolute values and as percentage of the total number of extensions per application for both case studies.

Application	Layer	generic								specific						Total
		System		Presentation		Application		Data		Presentation		Application		Data		
		#	%	#	%	#	%	#	%	#	%	#	%	#	%	#
Application	Time Recorder	3	10%	4	13%	1	3%	1	3%	10	32%	9	29%	3	10%	31
	Cross Compiler	3	8%	4	11%	1	3%	0	0%	9	24%	19	51%	1	3%	37
	Total	6	9%	8	12%	2	3%	1	2%	19	28%	28	40%	4	6%	68

Figure 7.6: Number and percentage of generic and specific extensions per application layer

For our evaluation, we did not only consider the case of building pure web applications, where all extensions are executed on a web server, but we also analyzed combinations of desktop and web applications, where the extensions in the application layer and in the data layer are executed on the server, but the extensions in the presentation layer are executed on the client-side computers. For architectures that combine server-side and client-side extensions, we distinguish between thin client applications and rich web applications. In thin client applications, the extensions in the presentation layer are deployed and executed on the client side, while in rich web applications those extensions are deployed on the server, but transferred to the client on demand and are executed in a sandbox there. For rich web applications, we used the Silverlight technology for implementing user interface extensions.

7.2.1 Prefabrication

When building web applications from existing desktop applications, some extensions cannot be reused as they are. For example, the discovery mechanism is implemented differently for single-user desktop applications and for multi-user web applications. Thus, the discovery extensions that are used by desktop applications need to be replaced by discovery extensions for the web. Similarly, extensions in the presentation layer are implemented differently for desktop and for web applications and need to be replaced as well. However, if those extensions are generic, they need not be re-implemented by the developer, but can be prefabricated and provided by the composition infrastructure. Thus, the degree of component prefabrication influences the degree of component reusability. The more components are already prefabricated, the higher is the degree of component reusability.

Figure 7.7 shows the number of prefabricated extensions and their percentage of the total number of extensions of our case study applications. Since all generic extensions could be prefabricated, 9 of 31 extensions were prefabricated for the *Time Recorder* application and 8 of 37 extensions were prefabricated for the *Cross Compiler* application. As a result, the average degree of component prefabrication was 25%, while 75% of all extensions had to be implemented exclusively for those applications. Furthermore, the figure shows that the number of prefabricated extensions in the system layer and in the presentation layer is significant higher as for the prefabricated extensions in the application layer and in the *data layer*.

		Layer									
		System		Presen-tation		Appli-cation		Data		**Total**	
		#	%	#	%	#	%	#	%	#	%
Appli-cation	Time Recorder	3	10%	4	13%	1	3%	1	3%	9	29%
	Cross Compiler	3	8%	4	11%	1	3%	0	0%	8	22%
	Total	6	9%	8	12%	2	3%	1	1%	17	25%

Figure 7.7: Number and percentage of prefabricated extensions per application layer

7.2.2 Reusability

Whether an extension can be reused both in desktop applications and in web applications depends on the environment in which an extension should be used (i.e., *Server*, *Client*, or *Sandbox*) and on the application layer to which an extension belongs. In other words, the architecture of an application influences the degree of component reusability. Thus, the degree of component reusability differs depending on whether a desktop application is adapted to a pure web application, to a thin client application, or to a rich web application. In addition to that, the amount of extensions in the various application layers also influences the degree of component reusability.

Figure 7.8 on the next page shows which extensions can be reused or need to be replaced in a certain environment depending on its application layer. The figure distinguishes between the environments *Server*, *Client*, and *Sandbox*. *Server* means that an extension is deployed and executed on the server; *Client* means that an extension is deployed and executed on the client; and *Sandbox* means that the extension is implemented as a Silverlight extension, which is deployed on the server, but executed in a sandbox on the client. Furthermore, the figure distinguishes between *reusing*, *replacing*, and *potentially replacing* an extension. *Reusing* means that an extension can be reused in a particular environment as it is; *replacing* means that the extension needs to be replaced either with a prefabricated extension, or with an extension that just needs to be re-compiled for this environment; and *potentially replacing* means that an extension may just needs to be recompiled for a particular environment depending on its implementation. Extensions that can be neither reused nor replaced must be re-

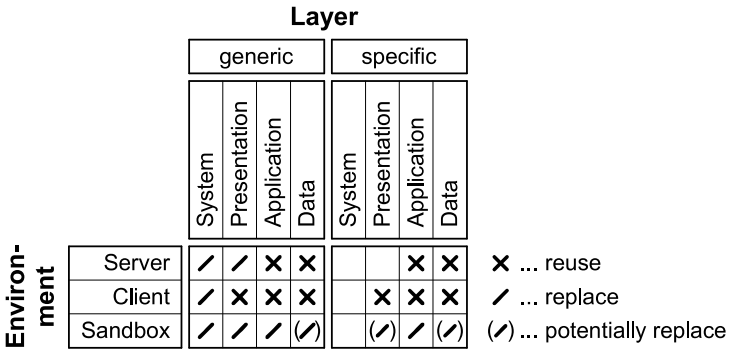


Figure 7.8: Reusability of generic and specific extensions depending on their deployment environment and their application layer

implemented for using them in the particular environment. Since the reusability of extensions was only dependent on the extension's deployment environment, its application layer, and whether it was generic or specific, the figure does not distinguish between both case studies and it does not distinguish between extensions in the same application layer for generic or for specific extensions.

In our case study the extensions in the system layer implement the discovery mechanism. Since the discovery mechanism is implemented differently in every environment, those extensions either need to be replaced by generic prefabricated discovery extensions for the respective environment, or those extensions need to be re-implemented for the respective environment if they are specific and implemented for a certain application. Since all discovery extensions in our case studies are generic extensions, they were provided by the composition infrastructure and it was not necessary to re-implement them. For the extensions in all other application layers, it depends on the deployment environment, whether they can be reused, whether they just need to be replaced, or whether they need to be re-implemented.

If extensions should be executed on the *Server*, they can be reused, if they belong to the application layer or to the data layer. Extensions in the presentation layer cannot be reused, because user interfaces for web applications are implemented differently than user interfaces for desktop applications. However, since generic extensions in the presentation layer are provided by the composition infrastructure, they just need to be replaced, while specific extensions in the presentation layer need to be re-implemented for using them in web applications.

If extensions should be used on the *Client*, they can be reused without modification in all application layers, except in the system layer. As mentioned above, system layer extensions need to be replaced or re-implemented. Even though the extensions in the presentation layer could be reused without modification, they were slightly adapted for performance reasons in our case studies. In the original implementation, the user interface was executed in the Plux runtime thread. Since this thread is used for distributed communication, communication delays would possibly have led to bad responsiveness of the user interface. Thus, we adapted those extensions so that the user

interface thread is now different from the runtime thread. However, after the adaption, these extensions can now be reused for desktop applications and for thin client applications without any further adaption.

Since *Sandbox* extensions are implemented with the Silverlight technology and Silverlight assemblies are not binary compatible to .NET assemblies, they cannot be reused as they are. However, generic extensions can be replaced with extensions provided by the composition infrastructure, and extensions in the application layer just need to be re-compiled and replaced, but not re-implemented. For extensions in the presentation layer it depends on the library that was used for implementing the user interface. If the user interface for a desktop application only used library elements that also exist in Silverlight, the extensions just need to be re-compiled, otherwise they need to be re-implemented. In our case studies, the user interface of the *Time Recorder* application was implemented with the intention to deploy it as a rich web application as well. Since it used only library elements that also exist in Silverlight, it was enough to re-compile and replace the user interface. However, the *Cross Compiler* was built with library elements that do not exist in Silverlight and thus they had to be re-implemented. Extensions in the data layer possibly need to be re-compiled as well, if they do not use local resources such as the local file system. In the case studies, the data layer extensions *Xml Data Store* as well as *Preferences* are using local resources and thus could not be replaced with re-compiled extensions. However, since it does not make sense to execute those data layer extensions in the *Sandbox* environment they were not re-implemented.

The following subsections show the degree of component reusability, which was achieved when adapting the case studies to pure web applications, to thin client applications and to rich web applications.

Pure Web Applications

In a pure web application every extension is executed on the web server and the application is accessed via a browser frontend. Thus, there are only *Server* extensions, but no *Client* and no *Sandbox* extensions. Figure 7.9 on the next page shows the degree of component reusability that could be achieved when adapting the desktop applications of our case studies to pure web applications.

The *Time Recorder* application consists of 31 extensions in total. We could reuse 14 extensions without modification, we replaced 3 generic extensions in the system layer and 4 in the presentation layer, and we had to re-implement 10 specific extensions in the presentation layer.

The *Cross Compiler* application consists of 37 extensions in total. We could reuse 21 extensions without modification, we replaced 3 generic extensions in the system layer and 4 in the presentation layer, and we had to re-implement 9 specific extensions in the presentation layer.

		Environment							
		Server		Client		Sandbox		**Total**	
		#	%	#	%	#	%	#	%
Time Recorder	reuse	14	45%	-	-	-	-	14	45%
	replace	7	23%	-	-	-	-	7	23%
	re-implement	10	32%	-	-	-	-	10	32%
Cross Compiler	reuse	21	57%	-	-	-	-	21	57%
	replace	7	19%	-	-	-	-	7	19%
	re-implement	9	24%	-	-	-	-	9	24%
Total	reuse	35	51%	-	-	-	-	35	51%
	replace	14	21%	-	-	-	-	14	21%
	re-implement	19	28%	-	-	-	-	19	28%

Figure 7.9: Component reusability for building pure *Web Applications*

As a result, 51% of all extensions could be reused, 21% could simply be replaced, and 28% needed to be re-implemented for the web. We consider it as a proof for the power and the location transparency of our plugin platform that only one third of the components had to be re-implemented when desktop applications were ported to the web.

Thin Client Applications

In thin client applications, the extensions that implement the user interface are deployed and executed on the *Client*, while the extensions that implement the business logic are deployed and executed on the *Server*. Figure 7.10 shows the degree of component reusability that could be achieved when adapting the desktop applications of our case studies to thin client applications.

		Environment							
		Server		Client		Sandbox		**Total**	
		#	%	#	%	#	%	#	%
Time Recorder	reuse	14	45%	14	45%	-	-	28	90%
	replace	3	10%	0	0%	-	-	3	10%
	re-implement	0	0%	0	0%	-	-	0	0%
Cross Compiler	reuse	21	57%	13	35%	-	-	34	92%
	replace	3	8%	0	0%	-	-	3	8%
	re-implement	0	0%	0	0%	-	-	0	0%
Total	reuse	35	51%	27	40%	-	-	62	91%
	replace	6	9%	0	0%	-	-	6	9%
	re-implement	0	0%	0	0%	-	-	0	0%

Figure 7.10: Component reusability for building *Thin Client Applications*

When adapting the *Time Recorder* desktop application to a thin client application we could reuse 14 extensions in the application and the data layer on the server as well as 14 extensions in the presentation layer on the client. Only the 3 discovery extensions in

the system layer needed to be replaced with the respective extensions for server-side and client-side discovery.

When adapting the *Cross Compiler* application we could reuse 21 extensions in the application layer and data layer on the server as well as 13 extensions in the presentation layer on the client. Similar to the *Time Recorder* application, only the 3 discovery extensions had to be replaced.

As a result, we could reuse 91% of all extensions and could replace 9% with other prefabricated extensions. Not a single component had to be re-implemented.

Rich Web Applications

In rich web applications, the extensions that implement the user interface are deployed on the *Server*, but are executed on the *Client*. The extensions that implement the business logic are deployed and executed on the *Server*. Figure 7.11 shows the degree of component reusability that could be achieved when adapting the desktop applications of our case studies to rich web applications.

		Environment							
		Server		Client		Sandbox		**Total**	
		#	%	#	%	#	%	#	%
Time Recorder	reuse	14	45%	-	-	0	0%	14	45%
	replace	3	10%	-	-	14	45%	17	55%
	re-implement	0	0%	-	-	0	0%	0	0%
Cross Compiler	reuse	21	57%	-	-	0	0%	21	57%
	replace	3	8%	-	-	4	11%	7	19%
	re-implement	0	0%	-	-	9	24%	9	24%
Total	reuse	35	51%	-	-	0	0%	35	51%
	replace	6	9%	-	-	18	26%	24	35%
	re-implement	0	0%	-	-	9	13%	9	13%

Figure 7.11: Component reusability for building *Rich Web Applications*

When adapting the *Time Recorder* desktop application to a rich web application, the reusability of business logic extensions on the server is the same as for both other architectures. Since the implementation of the user interface is done with Silverlight, extensions in the presentation layer cannot be reused. However, it was also not necessary to re-implement them, because they were implemented in such a way that they just had to be re-compiled for the Silverlight technology.

When adapting the *Cross Compiler*, all extensions in the application layer and the data layer could be reused. The discovery extensions in the system layer and the generic extensions in the presentation layer had to be replaced, but the specific extensions in the presentation layer had to be re-implemented, because they originally were implemented using the Windows Forms technology, which cannot be re-compiled for Silverlight.

In total, 51% of all extensions could be reused, 35% had to be replaced and only 13% had to be re-implemented. Again, this seems to be a strong demonstration of the power and location transparency of our plugin platform.

Summary

This chapter summarizes the contributions of this thesis, which consist of refinements of the original Plux component model and of contributions that enable distribution support, multi-user support, and web support. Furthermore, it points at some open issues that could not be resolved yet, and suggests a set of topics for further research in order to improve the concepts of the presented component model.

This thesis presents a component model for building plugin-based web applications and a component infrastructure that implements the component model. In contrast to existing solutions, the presented component model allows users to customize and extend their web applications with user-specific extensions. Furthermore, the component model supports distribution of components across multiple computers, which enables users to extend their web applications with client-side components that have access to local resources, such as local hardware, if required. The following sections conclude the thesis by summarizing research contributions, by pointing to open issues to be resolved, and by suggesting further research.

8.1 Contributions

The thesis claims research contributions for assembling web applications from plugins, so that every user can have his individual set of components and that components can reside on different computers. The thesis is based on the Plux component model of Wolfinger, which was described in his PhD thesis [Wolfinger, 2010]. However, it goes far beyond Wolfinger's thesis by introducing a number of refinements to the original component model and by extending it with new concepts that enable distribution support, multi-user support, and web support. The following subsections summarize these contributions.

Component Model Refinements

We introduced some refinements to the original Plux component model, which simplify extension development and lead to less coding effort as well as to less coding complexity:

- *New Meta-object Model.* The metadata for extensions can be retrieved via meta-objects (e.g., *Extension*, *Plug*, and *Slot*). Our new component model defines a completely redesigned meta-object model, which simplifies the composition API. The most significant modification is that in the original composition API developers had to distinguish between type meta-objects and instance meta-objects. In the new component model, developers only deal with instance meta-objects, while type meta-objects are used only internally by the composition infrastructure.

- *Synchronous Composition Process.* Our new component model defines a completely redesigned composition process. The original composition process was executing composition operations asynchronously, i.e., all composition operation were enqueued by the composer and extensions had to wait for callbacks to continue work after the operation was performed. The current composition process performs composition operations synchronously, i.e., when calls to composition operations return, one can be sure that they have already been executed. For this, the composition model introduced nested composition sequences. The synchronous composition process simplifies programmatic composition and leads to a more comprehensible implementations of extensions.

- *Lazy Composition / Automatic Garbage Collection.* Our new composition process activates extensions not until they are accessed for the first time. Since extensions only get composed as soon as they are activated, the composition process only composes extensions that are in use. Furthermore, the component model includes an automatic garbage collection mechanism, which destroys extensions, as soon as they are not used anymore. The combination of lazy composition and automatic garbage collection keeps the composition state minimal.

- *Enriched Composition State.* Our new component model allows tagging connections between extensions with an arbitrary number of named labels. In this way, the composition state can be enriched with additional information, e.g., by marking a contributor that is currently performing a certain task.

- *Additional Composition Events.* Our component model defines new composition events that notify extensions about upcoming composition operations and allow them to cancel them if required, e.g., if some precondition is not fulfilled.

- *Composition Behaviors.* Our new component model defines reusable composition behaviors, which implement common composition logic patterns and can be applied to extensions declaratively. Composition behaviors increase code reuse, because programmatic composition is extracted into reusable composition libraries that can be applied to different extensions.

- *Component Customization.* Our new component model defines a customization model for extensions by which extensions can be enriched with configurable settings. Settings for extensions can be retrieved via their meta-objects.

Component Model Extensions

The most important contributions of this thesis are the following novel concepts, which extend the original component model in order to support building plugin-based distributed multi-user web applications:

- *Distribution support*. Our extended component model defines a distributed discovery mechanism that allows users to install plugins on different computers without having to configure the application in a special way. It defines a distributed composition process that supports automatic composition of extensions that are located on different computers. It defines a distributed thread management that simulates a single coherent thread, which is assembled from multiple distributed threads that are linked together. Finally, it defines an interoperable interaction standard for distributed extensions that supports reference identity, data synchronization, and garbage collection for distributed objects.

- *Multi-user support.* Our extended component model defines a multi-user discovery mechanism that allows every user or user group to install their specific sets of plugins, which are kept in separate composition states. Thus, every user can compose an application independently from other users. Furthermore, our multi-user composition state uses separate memory areas for user-specific extensions so that errors in users-specific extensions do not affect other users.

- *Web support.* Our extended component model defines a deployment standard that allows users to host Plux web applications on a web server. It defines an exchangeable runtime thread that enables the web application to be executed in a different runtime thread per round trip.

8.2 Open Issues

There are two open issues that could not yet be resolved by this thesis. Both are caused by the transparent distribution of extensions across multiple computers. The first open issue is about disconnected distributed objects that are still referenced, while the second open issue is about complicated UI development for distributed extensions.

Disconnected Distributed Objects

In existing solutions, distribution support has to be programmed manually. Usually distribution applies to just a few specific software parts, which can be tested very well.

Plux, however, allows distributing any extension to other computers. Thus, minor programming mistakes may lead to errors that would not have occurred if an extension were used locally. One example for such a mistake, which actually occurred in practice, is not to unregister an event handler during decomposition.

As soon as a remote runtime node gets disconnected, all extensions that are living there get decomposed. However, if one of the decomposed extensions forgets to unregister one of its event handlers for an event of a remote extension, the next occurrence of this event will try to call the event handler, which does not exist any longer, so an exception will be thrown. Even though the notifier of the Plux composition infrastructure, which invokes all event handlers for composition events, can handle this situation, the same problem also occurs for events that are raised by extensions. Thus, all extensions possibly have to deal with disconnected event handlers. Furthermore, a disconnected event handler is just one example for this problem. The same problem can happen for every reference to a proxy for a distributed object that is kept after a distributed environment was disconnected.

This error can be avoided, if extensions are implemented in a clean way, so that all references to distributed objects get released, as soon as an extension gets decomposed. However, there is still a problem when an environment gets disconnected unexpectedly, e.g., because of losing the network connection. In this case, references to distributed objects cannot be released before the connection breaks. As a result, extensions that are distributed across multiple computers may be faced with unexpected exceptions caused by connection problems. Even though the Plux runtime can deal with unhandled exceptions, extensions that are not aware of this problem may get into an invalid state. In this case, the application needs to be restarted.

The problem of unexpected disconnection of distributed objects also exists in other technologies such as remoting, but those technologies have better opportunities to handle it, because distribution is implemented only in specific parts of an application.

Complicated UI Development for Thin Client Applications

The component model specifies a dedicated runtime thread, which is used for runtime operations as well as for communication between extensions. Since extensions can be distributed across multiple computers, it is a bad design to execute user interface operations in the runtime thread, because distributed communication may lead to delays and thus to bad responsiveness of the user interface. This is not a problem for web user interfaces, which are displayed in a web browser. It is only a problem for user interfaces that are implemented with an UI framework for desktop applications such as Windows Forms or WPF (e.g., the user interface of client-side extensions that are plugged into a web application). For such extensions, the UI thread should be different from the Plux runtime thread. If the user interface is executed in some other thread than the Plux runtime thread, extensions need to invoke method calls from the UI thread into the Plux runtime thread and the other way around. This has to be done asynchronously; otherwise there is no benefit of executing the user interface in a

different thread than the Plux runtime thread. This makes the implementation of user interfaces for distributed client-side extensions more complicated.

8.3 Future Work

There are ideas for improving the current component model. Some of the following topics are already work in progress.

Connection Recovery

The current component model does not define concepts for recovering connections in the case of unexpected connection losses. The component model could be extended with mechanisms that try to reconnect an unintentionally disconnected runtime node in order to avoid errors as described in the Section 8.2.

Persistence

In the current implementation of the composition infrastructure, a Plux runtime is started at the beginning of a web session and is kept alive until the end of the session. However, saving the current state of the runtime after each round trip and restoring it for the next round trip would reduce memory consumption on the server and would enable the use of server farms. The Plux composition library already implements a prototype for saving and restoring the composition state of the runtime, but an infrastructure that integrates this library into the server runtime is still missing.

Interoperability

The current Plux composition infrastructure is implemented under .NET. However, since the concepts of the component model are language independent, the composition infrastructure could also be implemented in other languages. This would allow users to connect .NET extensions with Java extensions, say. A Master's thesis [Spasov, 2013] already ported the base component model for desktop applications to Java. However, a port of the extended component model for the web is still missing.

Distributed Locking

The distributed interaction standard defines a distributed runtime thread that is implemented with multiple threads on different computers that simulate a single coherent thread. This works for executing code in a distributed thread, but the current implementation does not support a distributed locking mechanism. Therefore locks only work locally, but not on distributed components. A distributed locking mechanism would enable thread synchronization across computer boundaries.

Resource Constraints

Since users are allowed to install user-specific extensions on the web server, the current component model maintains separate memory areas for user-specific extensions in

order to avoid interference between extensions of different users. However, user-specific extensions that are executed on the server do not only increase the risk of executing error-prone extensions on the server, they also consume CPU and memory resources on the server. In order to avoid server congestion, resource constraints could limit CPU and memory consumption for user-specific extensions. Thus, users could only install a set of user-specific extensions that do not exceed the limit of resource consumption, which is permitted for a certain user. If user-specific extensions would exceed this limit, they could be automatically decomposed.

Debugging Support

In distributed applications, code is executed on different computers. Even though the distributed thread management simulates a single coherent thread across multiple computers, the code is still executed in different threads and method calls are transported from one thread to another via messages. As a result, debugging is difficult, because there is no continuous call stack. Additional debugging support for distributed threads would be helpful.

8.4 Conclusion

This thesis presented a novel approach for building plugin-based distributed multi-user web applications. It defined a component model that specifies a metadata standard that allows adding and removing plugins in a plug-and-play manner, a deployment standard that maintains local and remote plugins for individual users, a composition standard that connects independent plugin components seamlessly to a coherent web application, an interaction standard that enables local and distributed communication between plugin components, and a customization standard that maintains optional settings for plugins. The concepts that are presented in this thesis are validated with a composition infrastructure that implements the specifications in the component model as well as with case studies.

The composition infrastructure provides a platform that assembles user-specific web applications from plugins that are deployed locally on a single computer or distributed across multiple computers. Implementation transparency for distributed components allows developers to implement remotely connected components in the same way as locally connected components. This simplifies the implementation of distributed applications and allows reusing the same components on different environments. Thus, components can be reused to build different architectures for applications, such as pure desktop applications, thin client applications, pure web applications, and rich web applications. Furthermore, since the presented concepts are language independent, the composition infrastructure allows building applications that are composed from components that are developed with different technologies, for example from .NET components and from Java components.

Appendix A: Hosting Plux Web Applications

This chapter describes how to host a Plux web application within an ASP.NET web page by using a web control. It explains the structure of the virtual directory for an ASP.NET web application, the location of the various Plux assemblies that implement the composition infrastructure, and how to customize and extend the composition infrastructure for specific needs.

The current implementation of the Plux component model is realized with the .NET framework [Microsoft, 2012g]. Therefore, Plux web applications are hosted within ASP.NET [Microsoft, 2012k] web applications, which can be published, for example, with the Microsoft Internet Information Server (IIS) [Microsoft, 2013c]. An ASP.NET web application is stored in a virtual directory, which contains web pages, library assemblies, and other resources, such as images and configuration files. A Plux web application is accessed through a web page in the virtual directory. The web page contains a Plux web control, which starts the Plux server runtime and accesses it on subsequent web requests.

An ASP.NET virtual directory contains several predefined directories with special meanings. Figure A.1 on the next page shows an example of how these predefined directories are used to host a Plux web application in a virtual directory named *TimeRecorder*:

- The directory *App_Data* is used for storing data of any kind, e.g., database files or XML files. For example, the Plux *FileLogger* uses this directory as its default location for its log files.

- The directory *Bin* contains all assemblies that are used by an ASP.NET web application. These assemblies can either be precompiled ASP.NET web pages, or other assemblies that are referenced by web pages, such as the assembly *Plux.Web.AspNET.dll*, which contains the Plux web control. The Plux web control is used in ASP.NET web pages to accesses the Plux server runtime when the control is rendered (see Section A.1 below). The *Plux.Web.AspNET.dll* references the assemblies *Plux.dll*, *Plux.Web.dll*, and *Plux.Web.Server.dll*. *Plux.dll* implements the base component model and is used for Plux desktop applications and for Plux web applications; *Plux.Web.dll* implements the main

parts of the component model for the web, which are used in both, the server runtime and the client runtimes; and *Plux.Web.Server.dll* implements the composition infrastructure that is required for the server runtime. However, the *Bin* directory does not contain *contracts*, *plugins*, or user-specific library assemblies for Plux applications. These assemblies are stored in the *Repository* directory.

- The *ClientBin* directory contains the Silverlight assemblies of the web application, which are transferred to the client-side on demand, if a Silverlight plugin is composed. Beside the Silverlight assemblies *Plux.dll* and *Plux.Web.dll*, the *ClientBin* directory contains the assembly *Plux.Web.Client.dll*, which implements the composition infrastructure for the client runtime of the Silverlight environment.

- The *Repository* directory is the default directory for *contracts*, *plugins*, and libraries of Plux web applications. The structure of this directory is described in Section 5.2 Deployment Standard on page 100.

Furthermore, the virtual directory for the *TimeRecorder* web application contains the web page *TimeRecorder.aspx* with the Plux web control, and file *Web.config*, which contains the configuration of the ASP.NET web application, as well as the configuration of the Plux server runtime. The Plux web control and the configuration file are described in the following subsections.

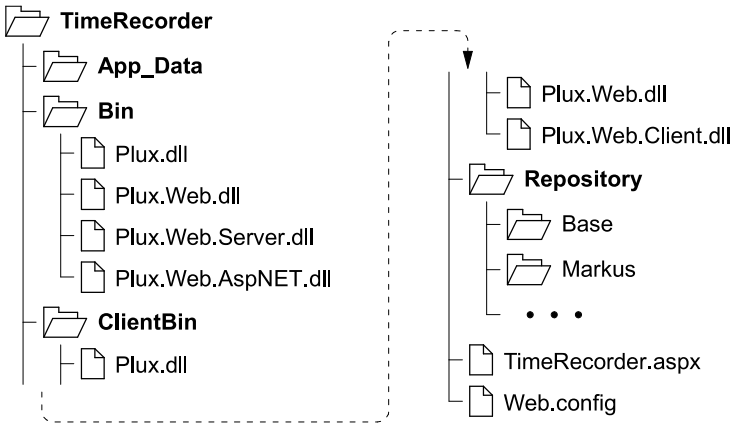


Figure A.1: Structure of the virtual directory for a Plux web application hosted with ASP.NET

A.1 The Plux Web Control

The Plux server runtime can be created and accessed either programmatically by code, or automatically by the Plux web control. This section describes how to include the Plux web control in an ASP.NET web page. Listing A.1 shows an ASP.NET web page

```
<%@ Page Language="C#" CodeBehind="TimeRecorder.aspx.cs" ... %>
<%@ Register Assembly="Plux.Web.AspNET" Namespace="Plux.Web"
    TagPrefix="plux" %>

<!DOCTYPE ... >
<html>
  <head>
    <title>Time Recorder</title>
  </head>
  <body>
    <form id="form1" runat="server" >
      <plux:Application runat="server" />
    </form>
  </body>
</html>
```

Listing A.1: Structure of an ASP.NET web page with a *Plux web control*

that includes the Plux web control. The web page starts with the directive *<%@ Page>*, which specifies, among other things, the language in which an associated code behind file is written and where to find the associated code behind file.

In order to make the Plux web control known to the ASP.NET page, the assembly *Plux.Web.AspNET.dll* has to be registered with the directive *<%@ Register>*. This directive specifies the assembly that implements the control, the control's namespace, and a tag prefix (*plux*) that is used to qualify the control in the web page. The Plux web control is included within an HTML *form* tag. The name of the control is *Application*, therefore it is included with the tag *<plux:Application runat="server"/>*, where *plux* is the registered prefix and *Application* is the class name of the web control. ASP.NET requires both tags to specify the attribute *runat="server"*. The web control can be included with an empty tag, as it is shown in Listing A.1, or it can contain configuration properties to customize the server runtime, e.g., to specify an application name or to set the location of the plugin repository. However, the server runtime can also be configured by a configuration file for the web application, as it is described in the following section.

When the web page is rendered for the first time, the web control creates the server runtime and instructs it to create a runtime node for the current session. Then it forwards the web request to the Plux application. The Plux application processes the web request and returns the result to the web page, which replaces the web control with the result in the rendered web response.

A.2 Runtime Configuration

The Plux runtime can be configured with various settings to customize it for the user's need. Furthermore, as the Plux runtime is built from exchangeable modules, it can be adapted and extended by further modules, where each module can have its individual

configurations, too. Configurations for the runtime and its modules can be specified programmatically with the runtime initializer or declaratively with a configuration file. As the component model is implemented with the .NET framework, the configuration for the runtime is specified in a *.NET Application Configuration File*, which is an XML file that is called *Web.config*.

Listing A.2 shows an example of a configuration file for the *Server Runtime*. In order to allow Plux configurations to be set and retrieved by the ASP.NET configuration model, they need to be declared in the <*configSections*> element of the configuration file. Thus, the configuration element <*section*> declares a Plux configuration section with its name and the type of the configuration section handler, which implements the configuration model of Plux. After the Plux configuration section has been declared, it can be used below and is enclosed within the <*plux*> configuration element.

The Plux configuration section consists of global configuration elements and module configuration elements. Module configuration elements are grouped into configurations for *core modules*, for *web modules*, and for *server modules*. Each of them specifies the type of the module, which is instantiated at start-up. The module type must implement the interface of the corresponding runtime module. Furthermore, module configuration elements can have sub elements to specify further configurations for the module. However, as all configuration elements are optional, module configuration elements can be omitted. In this case, the default type and the default configuration for the particular runtime module is used.

Values in the Plux configuration section can contain placeholders, which are enclosed within braces, as well as wildcards, which are indicated by asterisks. Placeholders and wildcards are replaced by concrete values when the configuration values are retrieved.

Plux specifies the placeholders *{user}*, *{group}*, *{application}*, and *{path}*. The placeholder *{user}* is replaced by the name of the user of the current session. The *{group}* placeholder

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="plux" type="Plux.Configuration, Plux.Web.Server"/>
  </configSections>
  <plux>
    <application name="TimeRecorder" company="ASE" version="0.1"/>
    <environmentUri value="plux://timerecorder.jku.at:25400" />
    <createAppDomain value="true" />
    <startupPaths>
      <add path="Repository/Base/Server/Plux.Web.Discoverer.dll" />
    </startupPaths>
    <arguments>
      <add key="imagePath" value="/Resources/Images" />
    </arguments>
```

```
    <coreModules>
      <dispatcher type="Plux.Web.Dispatcher, Plux.Web" />
      <loader>
        <add path="Repository/{group}/Server/**" />
      </loader>
      <logger>
        <add type="Plux.ConsoleLogger, Plux" verbosity="Normal"/>
        <add type="Plux.FileLogger, Plux" verbosity="Diagnostic"
             path="App_Data/Logs/{user}.log" />
      </logger>
    </coreModules>

    <webModules>
      <channel type="Plux.Web.TcpChannel" timeout="999" />
      <serializer>
        <formatters>
          <add type="System.Collections.Generic.List<>, mscorlib.dll"
               formatter="Plux.Web.ListFormatter, Plux.Web"/>
          ...
        <formatters/>
      </serializer>
    </webModules>

    <serverModules>
      <userStore type="Plux.Web.ConfigUserStore, Plux.Web">
        <add name="base" />
        <add name="ssw" parents="base" />
        <add name="mj" parents="ssw, base" />
      </userStore>
    </serverModules>

    <addons>
      <add type="..." />
    </addons>

  </plux>
</configuration>
```

Listing A.2: Structure of the configuration file *Web.config*

generates a collection of values, where the placeholder is replaced with the name of the current user and with all group names to which the user is a member. The *{application}* placeholder is replaced with the name of the application, which is set via the *Application* configuration element (see below). Finally, the *{path}* placeholder is replaced with the path element of the runtime's environment Uri (see below). The path of an Uri is the part between the authority and the query. For example the path of the Uri *plux://localhost:25400/TimeRecorder/Images?fullscreen=true* is */TimeRecorder/Images*. The example in Listing A.2 uses the placeholders *{user}* and *{group}*.

Wildcards are used in configuration values that specify a path. A single asterisk at the end of a path is replaced with a collection of paths containing all sub elements of the path. The double asterisk wildcard is replaced with a collection of paths that contains all sub elements of the path recursively. The example in Listing A.2 uses wildcards for

the loader paths, which are the paths to the directories from where assemblies can be loaded.

The following sub sections describe the configuration elements, which are shown in Listing A.2. Since many module configuration elements do not have further configuration options, they are omitted in the example.

Application

The <*application*> element specifies the application name, the company of the manufacturer, and the version of the application. The application information can be used to display it in the user interface. Furthermore, it is used in the connection string that is generated for connecting remote runtime nodes. Thus, multiple Plux web applications can be hosted within a single ASP.NET web page.

Environment Uri

The <*environmentUri*> element specifies an Uri that is used to connect remote runtime nodes to the server runtime. The listener of the server runtime accepts connections on the address of the environment Uri.

Create AppDomain

The <*createAppDomain*> element specifies whether the runtime should create separate AppDomains for user-specific plugins of different users. If the value is set to false, all plugins get loaded into the same AppDomain.

Startup Path

The <*startupPath*> element specifies the paths to the directories or files which should be initially discovered and composed at start-up time. In Listing A.2 the plugin *Plux.Web.Server.Discoverer.dll* is the plugin to be composed initially. As soon as this plugin is plugged to the Plux core, it discovers the other plugins for the current user.

Arguments

The <*arguments*> element specifies a collection of key value pairs, which can be retrieved by all extensions of the web application. It can be used, for example, to specify a path to a common resource.

Dispatcher

The <*dispatcher*> element specifies the type of the dispatcher module that should be instantiated for the runtime. This configuration element is empty, because the dispatcher has no further configurations. As the *Plux.Web.Dispatcher* is the default dispatcher type for each web runtime, this element could be omitted.

Loader

The <*loader*> element specifies the locations from which assemblies can be loaded. The example shows the path *Repository/{group}/Server/**. When this value is retrieved, the *{group}* placeholder is replaced with the name of the current user and with all group names of the user, e.g., if the user *Markus* is the only member of the group *Base*, the path is resolved to the paths *Repository/Markus/Server/** and *Repository/Base/Server/**. The wildcard at the end of the path specifies that the assemblies in all recursive sub directories can be loaded, too. Thus, the assemblies in the *Server* directory can be organized into sub directories, for example into the directories *Contracts*, *Plugins*, and *Libraries*. As there is no type specified in the <*loader*> element, the default loader is used.

Logger

The <*logger*> element specifies which logger should be used by the runtime. This element allows registering multiple loggers at the same time. For this, multiple logger types can be set by sub configuration elements. The configuration file of the example above specifies two loggers: the *ConsoleLogger*, which writes its log messages to a console window, and the *FileLogger*, which writes its log messages into a log file. Each logger can have a verbosity level that specifies the detail level for the messages to be logged. Furthermore, some loggers, such as the *FileLogger*, specify a path that is used as a target location for the log messages. The *FileLogger* in the example uses a separate log file per user, which is specified by the *{user}* placeholder.

Channel

The <*channel*> element specifies which channel type is supported by the runtime to communicate with remote runtime nodes. Similar to the configuration for the logger, the <*channel*> element can specify multiple supported channels. All supported channels are provided to remote environments when a connection is established. Remote environments can thus choose the channel type for communication. The timeout attribute specifies the amount of time the channel will wait until it raises an exception.

Serializer

The <*serializer*> element specifies an optional serializer type as well as the formatters that should be used for object serialization. Since the type of an object defines the formatter to be used, formatters are registered with the object type as their key, and the formatter type as their value. All registered formatters need to be available on each connected runtime node.

User Store

The <*userStore*> element specifies the type of the user store to be used. The *Plux.Web.ConfigUserStore* is a simple user store that can be used during application

development. It allows specifying users and their groups directly in the configuration file just by adding their names and their parents as sub configuration elements.

Add-ons

The <*addons*> element is used to register runtime add-ons that should be included by the runtime. Sub elements specify the type of the add-on, which gets instantiated at start-up time and which connects itself to the hooks of runtime modules when it gets initialized.

Appendix B: Runtime Procedures

The component model specifies operations and processes, the implementations of which are not trivial. This appendix provides a number of sequence diagrams that describe runtime procedures, which implement the specified operations and processes.

B.1 Runtime Lifetime

The lifetime of the runtime comprises three phases: *Startup*, *Run*, and *Shutdown*. Each phase differs between the server runtime and the client runtime. The *Startup* phase and the *Shutdown* phase are described in an individual sequence diagram for both, the server runtime and the client runtime. The *Run* phase is described in a single sequence diagram, but has a parameter to differ the behavior between the server runtime and the client runtime.

B.1.1 Startup

The Plux *Runtime* is created and configured by an *Initializer*, which reads the configurations from the runtime settings file (see Section A.2 Runtime Configuration). First the initializer creates the runtime with its runtime modules (see Section 6.1 Composition Infrastructure). When the runtime is created, its runtime state is *Created*, its dispatcher is *Released*, and the coordinator does not have the *Token* and it is *Idle*. The state *Idle* indicates that the runtime thread is currently not running, i.e., the dispatcher either is released, or the runtime thread is waiting for a new dispatcher operation to be executed. Now the runtime can be initialized and started.

Server Runtime

The server runtime is started via the *Server Initializer*. Its *Start* method calls the coordinators *Init* method with the token as an argument. In *Init*, the coordinator acquires the dispatcher, sets the token, and registers the dispatcher event handlers *Acquiring*, *Acquired*, *OpEnqueuing*, *OpEnqueued*, *OpFinished*, *Releasing*, and *Released*. These event handlers are used for thread management and communication coordination; their implementation is described in the following sections below. As *Init* acquires the dispatcher, it returns in the runtime thread *1.0*. Thus, the runtime thread

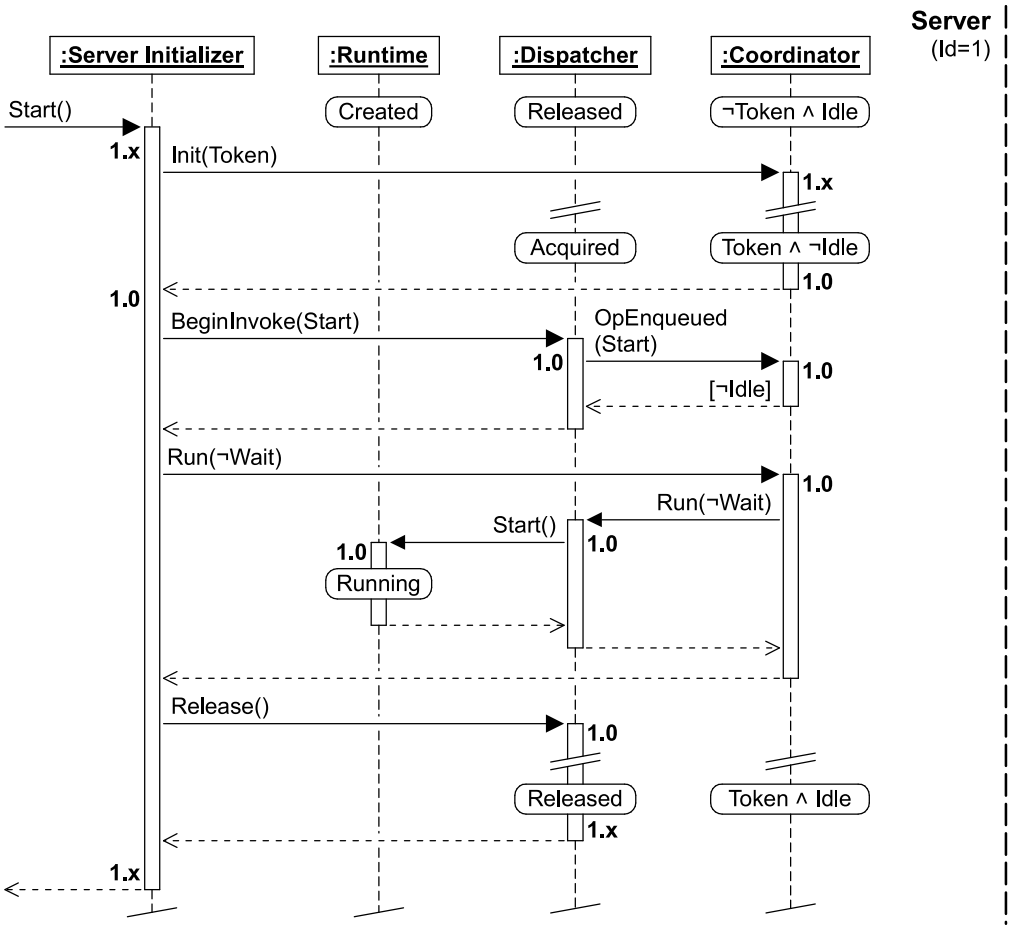


Figure B.1: Starting the server runtime

now is executing and the coordinator is *Not Idle*. After initializing the coordinator, it is ready for thread management and for coordinating the communication with connected environments.

Next, the initializer asynchronously invokes the first dispatcher operation to be executed when the dispatcher is started, which is the *Start* method of the runtime. As the dispatcher is *Acquired* and as the coordinator has the *Token* and is *Not Idle*, the *OpEnqueued* event handler has nothing to do in this case (see Section B.2.3 below).

Now the initializer calls the coordinator's *Run* method with *Not Wait* as an argument, which indicates that the *Run* method should return after all dispatcher operations are finished. The coordinator's *Run* method (see Section B.1.2 below) starts the dispatcher by calling the dispatcher's *Run* method, again with *Not Wait* as an argument. The dispatcher executes the enqueued dispatcher operation *Start*, which performs bootstrap discovery and the initial composition of the Plux application. After starting the runtime, its runtime state is *Running*. As soon as the dispatcher's *Run* method returns, the coordinator's *Run* method returns too, and the runtime is started.

As the server runtime does not block the runtime thread, but acquires the dispatcher on demand when it needs to execute code in the runtime thread, the initializer finally releases the dispatcher. When the initializer's *Start* method returns, the runtime is *Running*, the dispatcher is *Released*, and the coordinator has the *Token* and is *Idle*.

Client Runtime

The client runtime is started via the *Client Initializer*. The *Client Initializer*'s *Start* method starts a new thread, which is used as runtime thread for the whole runtime lifetime. After the initializer's *Run* method is started in the new thread, the *Start* method waits until the client runtime is connected. *Run* now calls the coordinator's *Connect* method, which connects the runtime to the server runtime. Additionally, *Connect* initializes the

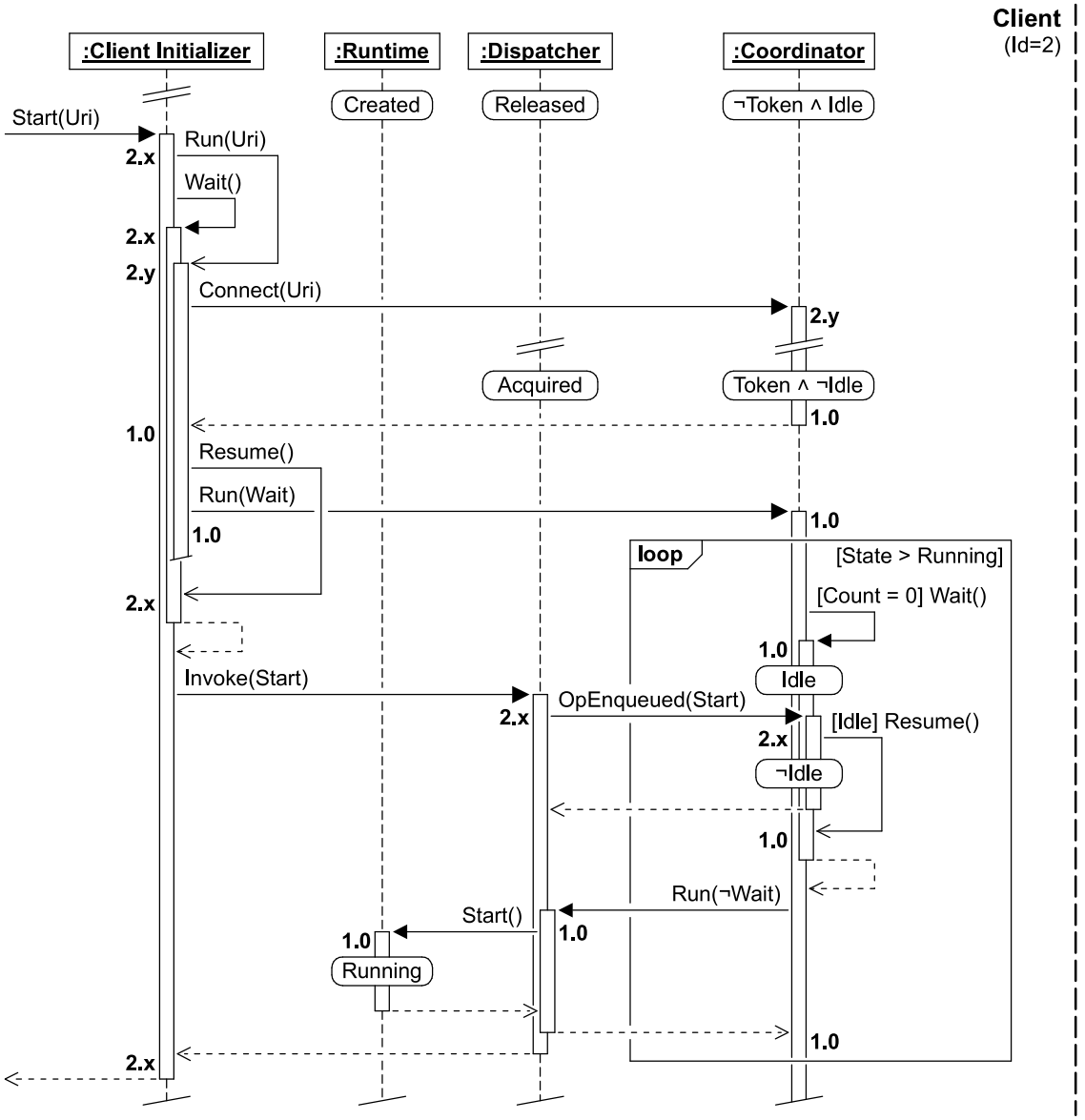


Figure B.2: Starting the client runtime

coordinator. It registers the event handlers, *Acquiring*, *Acquired*, *OpEnqueuing*, *OpEnqueued*, *OpFinished*, *Releasing*, and *Released*, requests the token, and acquires the dispatcher. *Connect* returns in the runtime thread *1.0*; the dispatcher is *Acquired*, and the coordinator has the *Token* and is *Not Idle*.

As soon as the runtime is connected, the initializer resumes its waiting *Start* method and calls the coordinators *Run* method with *Wait* as an argument, which indicates that the coordinator should block the runtime thread, when it is idle. The coordinator's *Run* method checks whether the dispatcher has any enqueued dispatcher operations. If not, it blocks the runtime thread and sets its state to *Idle*. Otherwise, it would call the dispatcher's *Run* method to execute the enqueued operations.

After the *Client Initializer*'s *Start* method is resumed, it synchronously invokes the runtime's *Start* method, i.e., the dispatcher's *Invoke* method returns after executing *Start* is finished. When the coordinator receives the *OpEnqueued* event, it resumes the runtime thread and sets its state to *Not Idle*, if it was *Idle*. The runtime thread continues and calls the dispatcher's *Run* method with *Not Wait* as an argument. The dispatcher executes the runtime's *Start* method, which bootstrap discovers the client-side plugins and plugs them to the connected server-side application. After that, the client runtime is *Running*, the dispatcher's *Run* method returns to the coordinator, and the *Invoke* method returns to the initializer. The coordinator again blocks the runtime thread, until a new dispatcher operation is enqueued. The initializer now returns the *Start* method and the runtime is started.

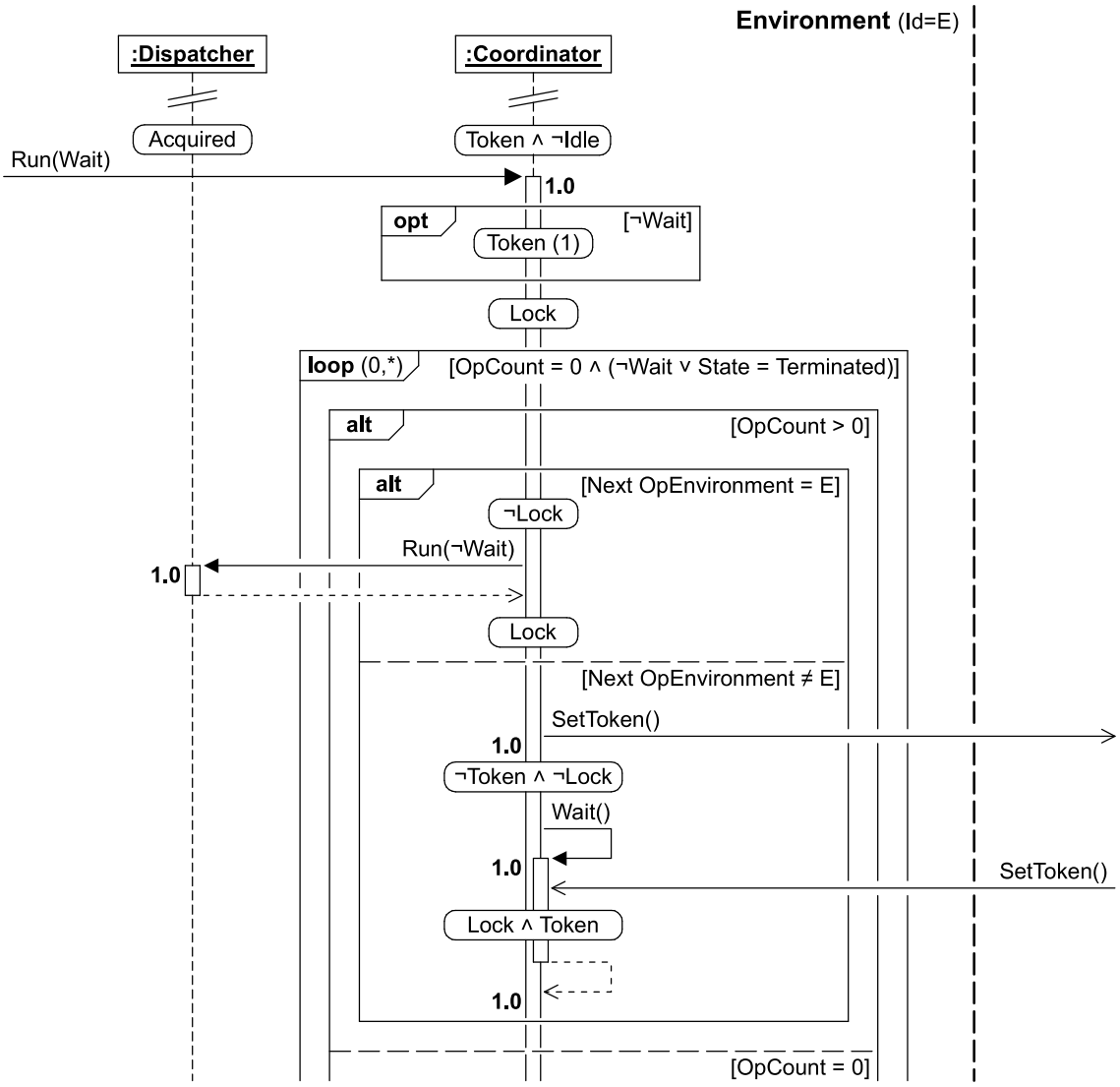
B.1.2 Run

The coordinator's *Run* method implements a main part of the coordinator's thread management. It starts the dispatcher, executes enqueued dispatcher operation, and passes the token to connected runtime nodes, if they need them to continue executing in the runtime thread there. Its parameter *Wait* defines whether the method should return after the dispatcher's operation queue is empty, or if it should block the runtime thread until a new dispatcher operation is enqueued. *Run* always must be called within the runtime thread *1.0*. Thus, at the beginning the dispatcher must be *Acquired*, the coordinator needs to have the *Token* and it is *Not Idle*.

If *Run* should not wait after the dispatcher is finished, it adds the current environment id to the token terminus. The token terminus is a list of environments to which the token is sent, as soon as the runtime thread is idle. This is necessary for all environments that do not block the runtime thread, but release the dispatcher when the runtime thread is idle. Otherwise, if the last dispatcher operation is executed on a remote environment, this environment would keep the token until it receives a new dispatcher operation from another environment. As the current environment needs the token to continue executing the runtime thread in order to be able to release the dispatcher, it needs the token after the dispatcher idle.

As *Run* needs to check the state of the dispatcher, which is changed from different threads, the coordinator uses a *Lock* object, which is set and released when the dispatcher state is retrieved and when the dispatcher state is modified. *Run* enters a loop, if either the dispatchers operation queue is not empty, or *Run* is waiting for new dispatcher operations if the dispatcher is idle, and the runtime state is not *Terminated*. The body of the loop behaves different whether the operation queue is empty, or not.

If the operation queue is not empty, the coordinator checks whether the next dispatcher operation was enqueued from the local environment with the id *E*, or from a remote environment. If it is a local dispatcher operation, the coordinator starts the dispatcher by calling the dispatcher's *Run* method. If it is a remote operation, it does not start the local dispatcher, but sends to token to the remote environment. This continues executing the runtime thread there and thus the remote dispatcher operation gets executed on the remote environment. After sending the token, the current environment does not have the token anymore and blocks the runtime thread until it



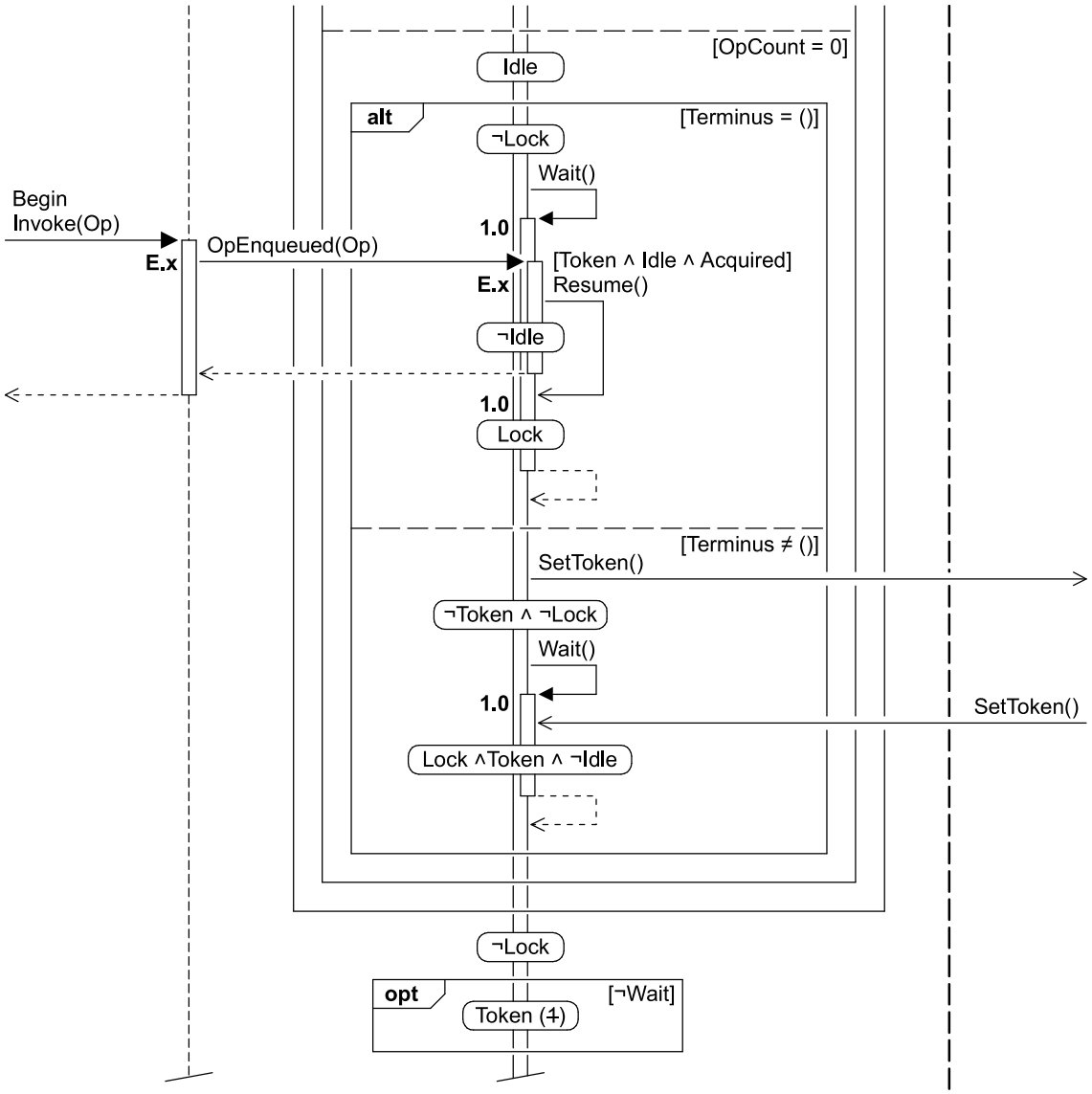


Figure B.3: Running the coordinator

receives the token again. This will happen because either a further dispatcher operation needs to be executed in the local environment, or because all dispatcher operations are finished and the token terminus was set to the local environment.

If the operation queue is empty (at the beginning of the loop), then it is guaranteed that *Run* was called with *Wait* as an argument and the runtime state is *Running*. Otherwise, the loop would not have been entered. In this case, the runtime thread is *Idle* and thus checks whether the token terminus is set to any environment.

If no token terminus is set, the coordinator blocks the runtime thread until a new dispatcher operation is enqueued. This can be done from another local thread, or from a remote environment with the *Invoke* communication operation (see Section 5.4.3). When the coordinator handles the *OpEnqueued* event, it resumes the runtime thread, if

the dispatcher is *Acquired*, and the coordinator has the *Token* and it is *Idle* (see Section B.2.3 below). Afterwards the loop continues at the beginning and starts executing the enqueued operation.

If the token terminus is set to a remote environment, the coordinator sends the token to the according environment and blocks the runtime thread, because it does not have the token anymore. As soon as the token is replied to the current environment, e.g., because a local thread enqueued a dispatcher operation, the coordinator resumes the local runtime thread and continues execution.

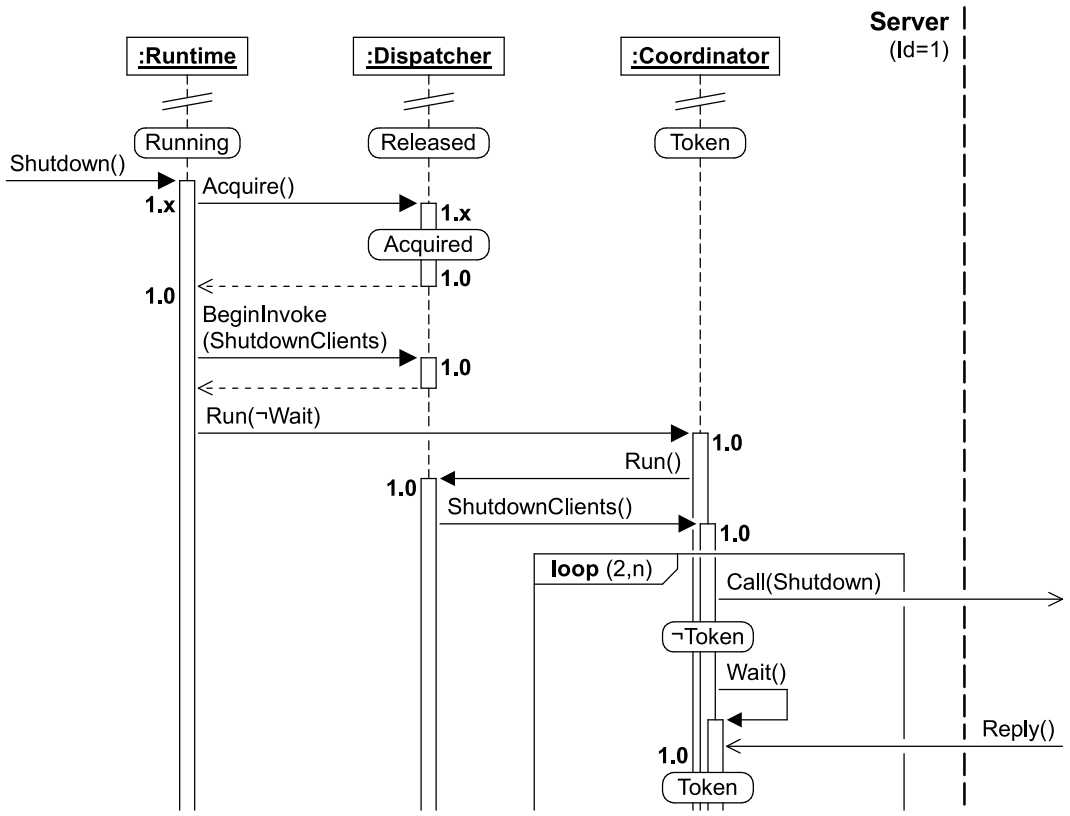
When *Run* exits its loop, because the dispatcher's operation queue is empty and *Run* is not waiting, *Run* removes the token terminus from the token and returns to its caller.

B.1.3 Shutdown

The *Shutdown* phase decomposes the application and disconnects runtime nodes. The following subsections describe the *Shutdown* phase for the server runtime and the client runtime.

Server Runtime

Shutdown on the server runtime shuts down all client runtimes too. The runtime method *Shutdown* first acquires the dispatcher and asynchronously enqueues the dispatcher operation *ShutdownClients*. Afterwards, it calls the coordinator's *Run*



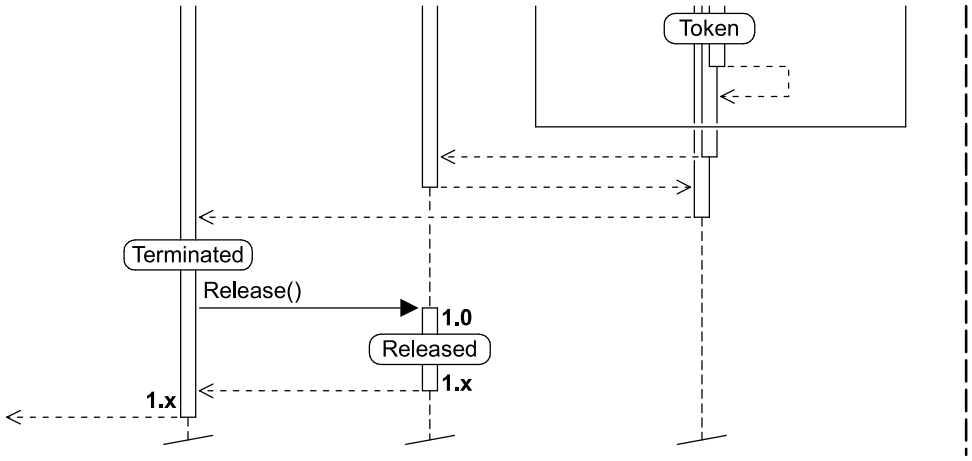


Figure B.4: Shutting down the server runtime

method, which starts the dispatcher. Thus, the dispatcher executes the coordinator's method *ShutdownClients*, which iterates over all connected environments and sends a *Call* operation with the runtime's *Shutdown* method as an argument to all of them. This causes that all clients decompose their extensions, remove all their plugins, and finally disconnect themselves (see Shutdown Client Runtime below). When the dispatcher has finished all dispatcher operations, all clients are terminated and disconnected. Now the server runtime decomposes all its extensions, removes its plugins, and stets its runtime state to *Terminated*. Finally *Shutdown* releases the dispatcher and returns to its caller.

Client Runtime

Shutdown on the client runtime decomposes and removes all client-side plugins and disconnects the runtime node from all its connected environments. The runtime's *Shutdown* method is either called by the server, if the server runtime is terminating, or it is enqueued as dispatcher operation on the client-side, as it is shown in the figure below.

During executing *Shutdown*, the runtime decomposes and removes all client-side plugins, sets its runtime status to *Terminated* and asynchronously invokes the coordinator's *Disconnect* method. *Shutdown* does not call *Disconnect* immediately, because at the time when *Shutdown* is executed, some client-side extensions may already have enqueued some dispatcher operation before *Shutdown* was called, or some extensions may enqueue a dispatcher operation during their termination. As soon as the runtime state is *Terminated*, client-side code is not allowed anymore to enqueue new dispatcher operations (see Section B.2.3 below). Thus, *Disconnect* is the last dispatcher operation that is executed on the *Client* environment.

The *Disconnect* method iterates over all connected environments and sends the *Disconnect* operation to them. The *Server* is the last environment to which the *Disconnect* operation is sent. The last *Disconnect* operation instructs the Server via an argument, to keep the token even though the *Reply* message is sent within the runtime thread. After

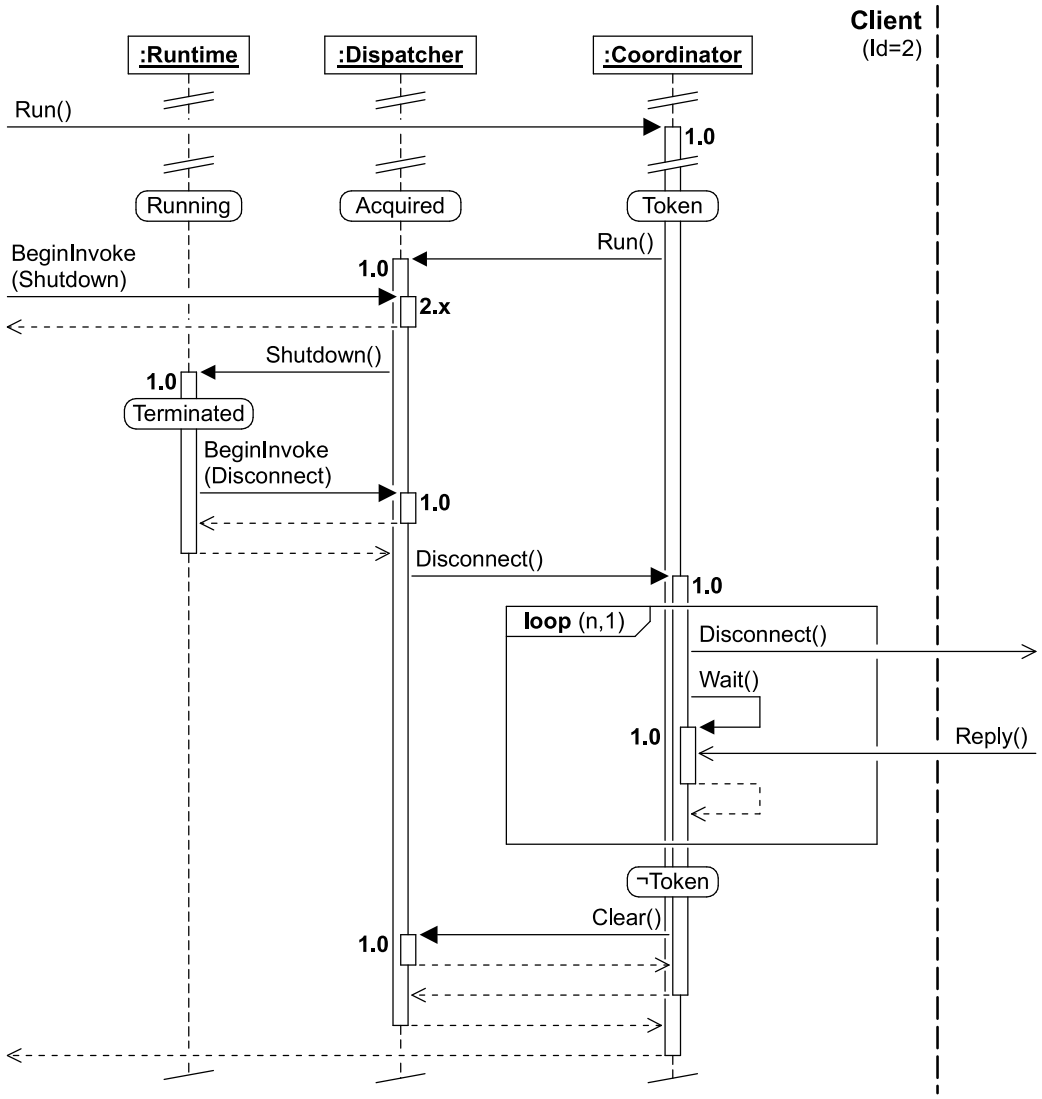


Figure B.5: Shutting down the client runtime

every runtime node is disconnected, the coordinator does not have the token anymore and removes all remaining dispatcher operations from the dispatcher queue by calling the dispatcher's *Clear* method. Before that, there may are some remote dispatcher operations in the queue, which still will be executed, because the token was passed to the *Server* environment. After the dispatcher queue is empty, the dispatcher's *Run* method returns to the coordinator. As the runtime state now is *Terminated*, the coordinator's *Run* method returns too, by which finally the runtime thread terminates.

B.2 Dispatcher Operations

The dispatcher is used to invoke method calls from any thread in the runtime thread. In order that the coordinator can coordinate the distributed runtime thread it needs to handle certain of dispatcher events, which are raised when the dispatcher gets

acquired and released, or when dispatcher operations get enqueued. The following subsections describe the implementations of these event handlers.

B.2.1 Acquire

Acquire is used to set the executing thread to the dispatcher thread, i.e., to the runtime thread. *Acquire* raises an *Acquiring* event just before the dispatcher changes its state to *Acquired* and an *Acquired* event just after it has changed its state. As the coordinator checks in some methods whether the dispatcher currently is *Acquired* or not, the coordinator requests the *Lock* object in the *Acquiring* event handler in order that the dispatcher's *Acquired* state cannot be changed while the coordinator is performing any other operation in which it currently holds the *Lock* object.

As soon as the dispatcher is *Acquired*, the runtime thread is not idle anymore and thus the coordinator changes its state to *Not Idle*. Since the coordinator needs the token to be

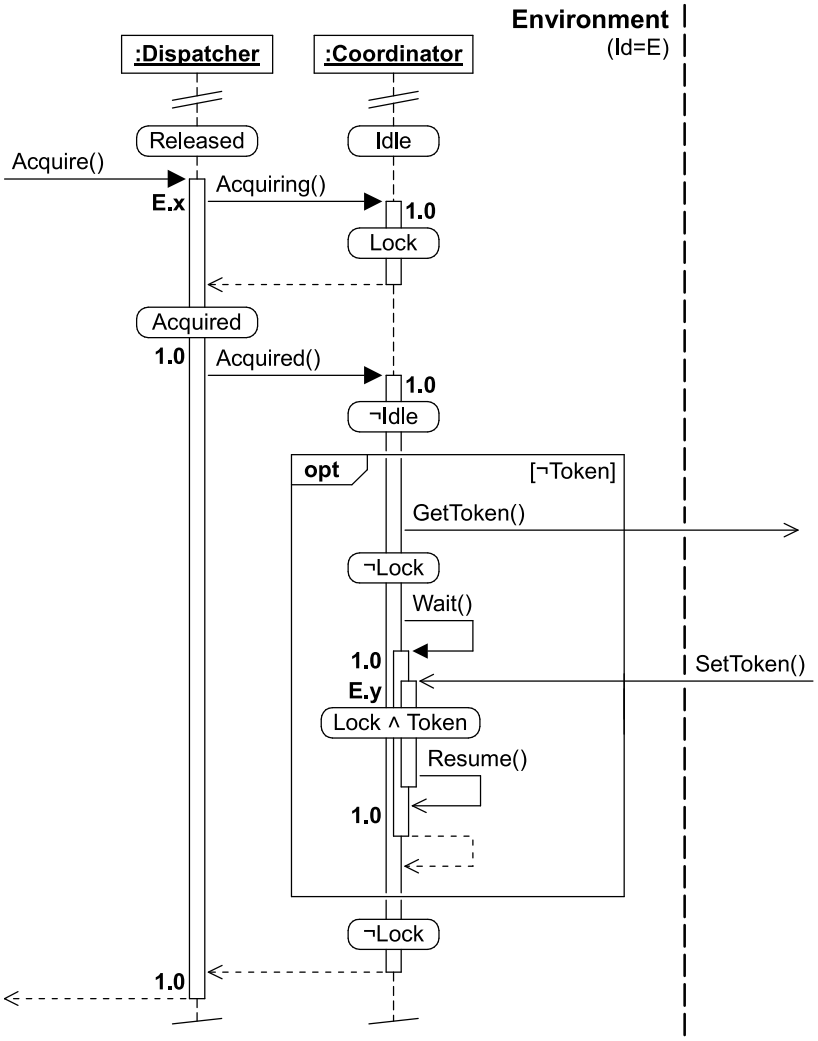


Figure B.6: Acquiring the dispatcher

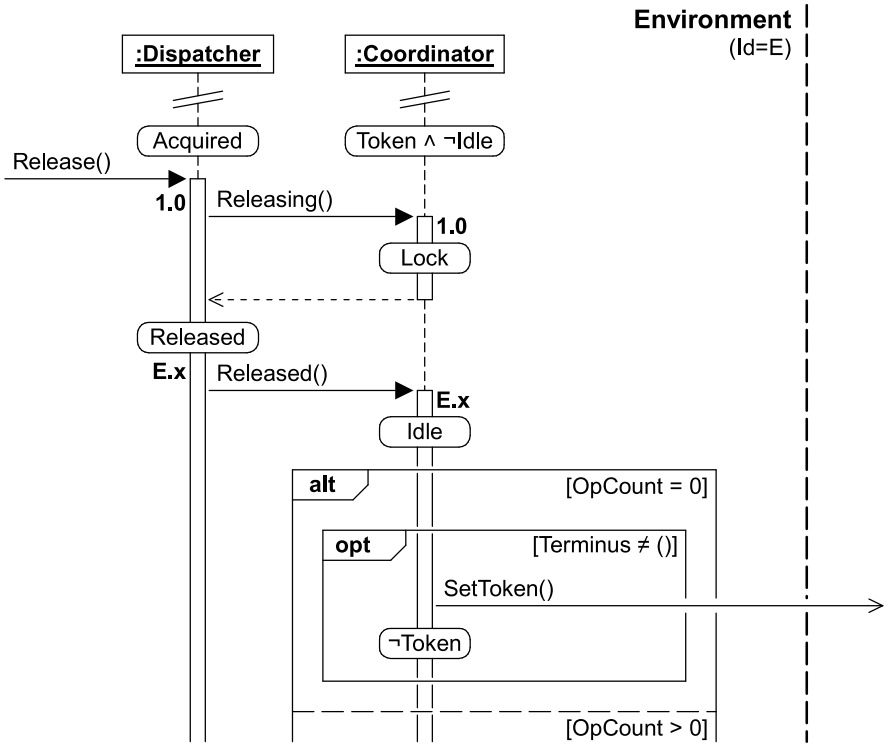
allowed to execute code in the runtime thread, the coordinator checks whether the local environment currently has the token. If not, the coordinator requests the token by sending the *GetToken* operation to its connected environments and waits until it receives the token. Every time before the coordinator is waiting, it releases its *Lock* object in order that other operations in other threads can be executed while the current thread is waiting.

When the coordinator receives the token, it again requests its *Lock* object and resumes the waiting runtime thread as the local environment now has the token. Finally the coordinator releases the *Lock* object and the local environment can continue execution in the runtime thread.

B.2.2 Release

Release is used to unset the current thread as dispatcher thread, i.e., as runtime thread. The dispatcher gets released when the runtime thread is idle, but the runtime thread should not be blocked. Release only can be called in the runtime thread. Since *Release* is only allowed to be called in the runtime thread, it is guaranteed that the coordinator currently has the token and the state of the coordinator is *Not Idle*.

The event handler *Releasing* requests the *Lock* object so that the dispatcher's *Acquired* state does not change, while other threads check the state of the dispatcher. After the dispatcher is released, it calls the *Released* event handler, which sets the state of the coordinator to *Idle* and checks whether dispatcher's operation queue is empty.



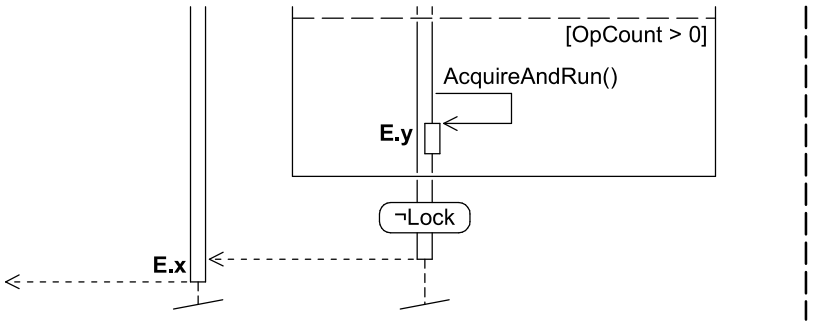


Figure B.7: Releasing the dispatcher

If the dispatcher's operation queue is empty, the coordinator checks whether a token terminus is set. If so, it sends the token to the environment that is set as token terminus.

If the dispatcher's operation queue is not empty, then the remaining dispatcher operations need to be executed. Thus, the coordinator calls *AcquireAndRun*, which uses another thread to acquire the dispatcher, call the coordinator's *Run* method, and release the dispatcher afterwards (see *AcquireAndRun* below).

Finally, the *Released* event handler releases the coordinator's *Lock* object.

AcquireAndRun

AcqurieAndRun acquires the dispatcher, calls the coordinator's *Run* method with *NotWait* as an argument, and releases the dispatcher again. This method is used when the dispatcher gets released, while there are still dispatcher operations in the operation queue. *AcquireAndRun* also is used when a dispatcher operation gets enqueued, while the dispatcher is released, but the coordinator has the token (see Section B.2.3 below).

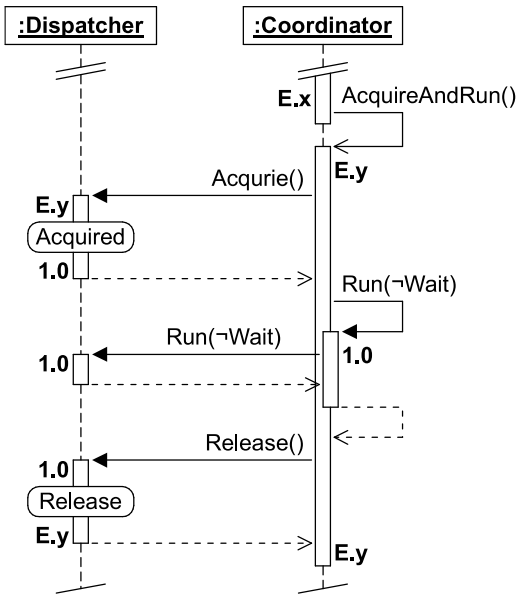


Figure B.8: Acquire and run the dispatcher to empty the operation queue

B.2.3 Invoke and BeginInvoke

Invoke and *BeginInvoke* are used to enqueue an operation in the dispatcher's operation queue in order to start the execution of a method from any thread in the dispatcher thread. The difference between *Invoke* and *BeginInvoke* is that *Invoke* does not return until the operation is executed, while *BeginInvoke* returns just after the operation was enqueued. The following figures only show the *BeginInvoke* method to describe the event handlers for the different dispatcher events. However, the dispatcher events for *Invoke* are the same. As soon as a dispatcher operation is enqueued in any thread, the dispatcher raises the *OpEnqueuing* event just before it enqueues the operation and the coordinator requests its *Lock* object, because the state of the dispatcher is about to be changed.

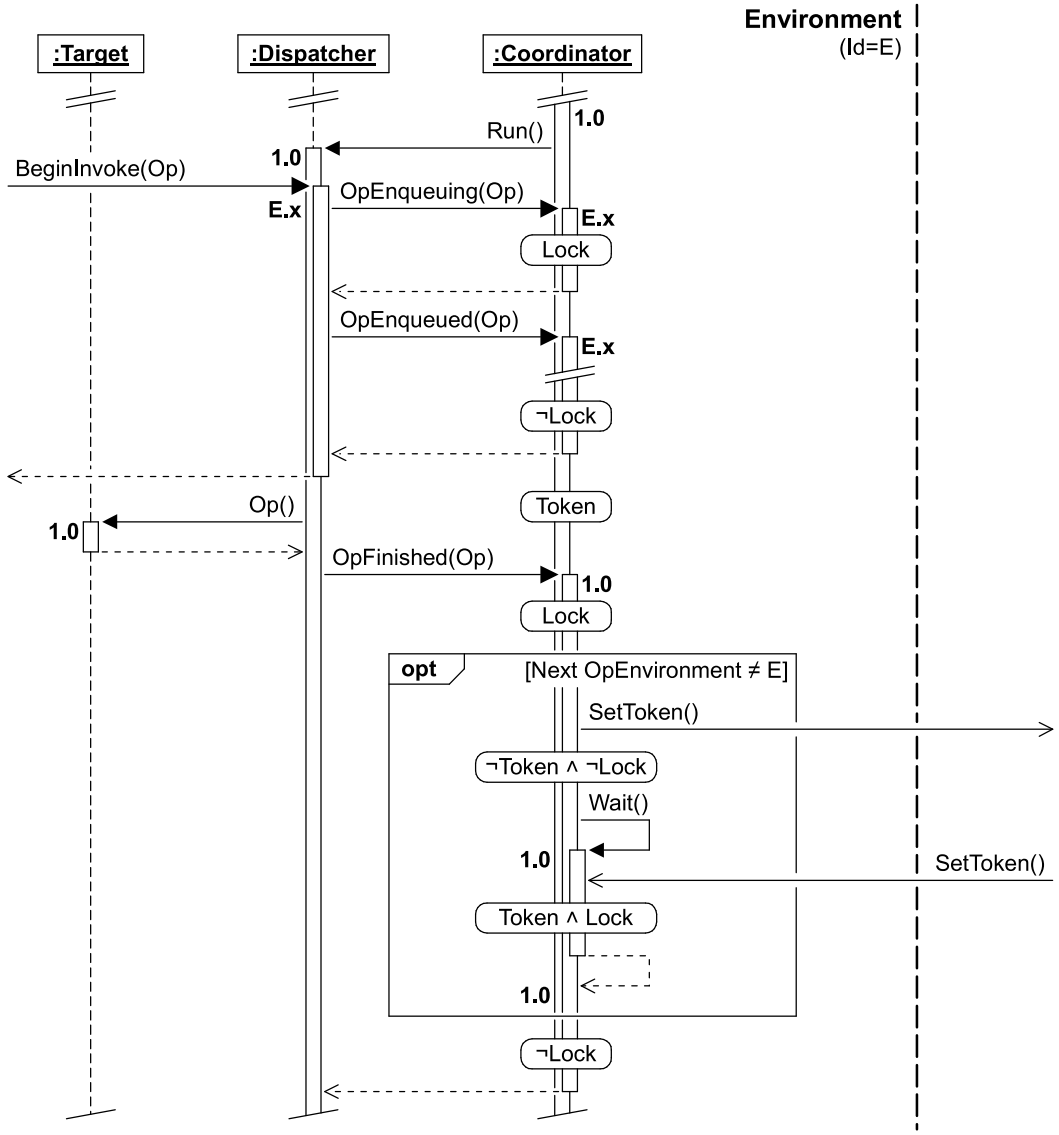


Figure B.9: Enqueuing a dispatcher operation

An operation only can be enqueued, if the runtime state is not *Terminated*. If the runtime already is *Terminated*, the *OpEnqueuing* event handler throws an exception (see subsection below).

When the dispatcher operation is enqueued, the coordinator handles *OpEnqueued* event, which behaves different depending on the current state of the dispatcher and the state of the coordinator. Since there are many different cases for the *OpEnqueued* event handler, the different cases are described in the subsections below. At the end of *OpEnqueued*, the coordinator releases the *Lock* object.

After the execution of the operation is finished, the dispatcher raises the *OpFinished* event. The event handler again requests the *Lock* object and checks whether the next dispatcher operation was enqueued from a remote environment. If so, it sends the token to this environment to continue executing there and waits until the local environment receives the token again. Before the coordinator calls *Wait*, it releases the *Lock* object. As soon as the token is received again, the *Lock* object is requested and the runtime thread continues executing. At the end of *OpFinished*, the coordinator releases its *Lock* object.

Invoke and BeginInvoke (Terminated)

It is not allowed to enqueue new dispatcher operations using *Invoke* or *BeginInvoke* if the runtime state is *Terminated*. In this case, the *OpEnqueuing* event handler throws an exception before the dispatcher enqueues the operation in its operation queue.

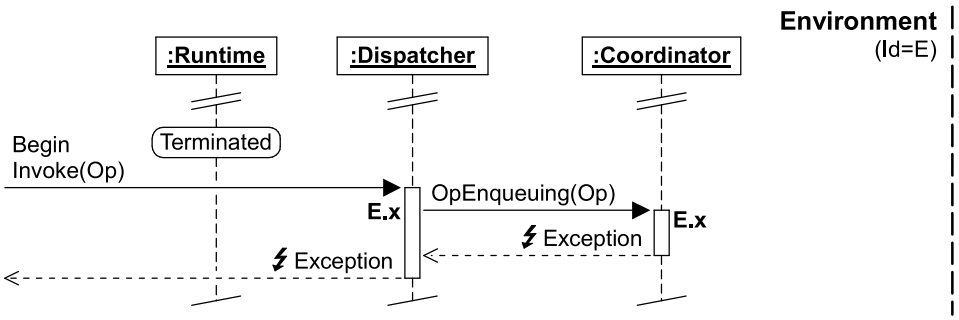


Figure B.10: *OpEnqueuing* event handler (*Terminated*)

OpEnqueued (¬Token)

When a dispatcher operation was enqueued while the coordinator does not have the token, the operation queue of the local dispatcher does not have a valid state. Thus, the operation needs to be sent to the environment that has the token. For this, the event handler uses the *Invoke* communication operation (see Section 5.4.3) with the dispatcher operation as an argument.

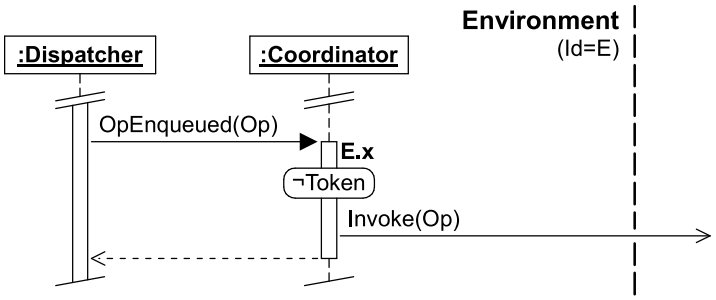


Figure B.11: *OpEnqueued* event handler (¬*Token*)

OpEnqueued (Token ∧ ¬Idle)

When a dispatcher operation was enqueued while the coordinator has the token and its state is *Not Idle*, the *OpEnqueued* event handler has nothing to do.

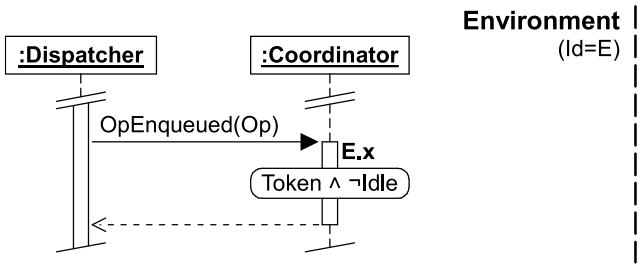


Figure B.12: *OpEnqueued* event handler (*Token* ∧ ¬*Idle*)

OpEnqueued (Token ∧ Idle ∧ Acquired)

When a dispatcher operation was enqueued while the coordinator has the token, its state is *Idle*, and the dispatcher is *Acquired*, it is guaranteed that the runtime thread currently is waiting. In this case, the coordinator resumes the runtime thread and sets its state to *Not Idle*.

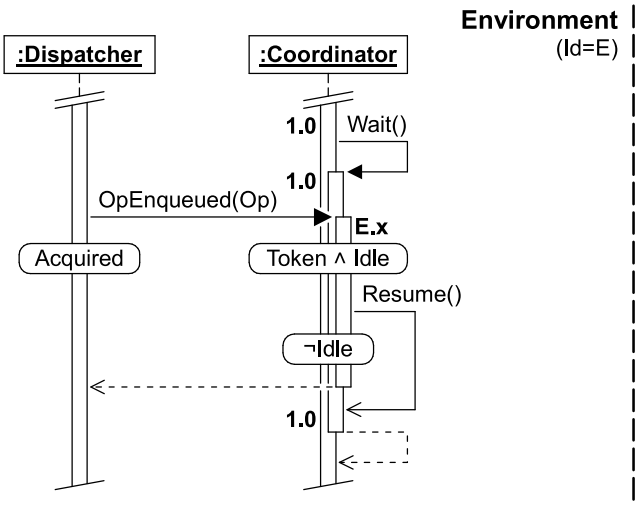


Figure B.13: *OpEnqueued* event handler (*Token* ∧ *Idle* ∧ *Acquired*)

OpEnqueued (Token ∧ Idle ∧ Released)

When a dispatcher operation was enqueued while the coordinator has the token, its state is *Idle*, and the dispatcher is *Released*, the coordinator checks whether the operation was enqueued from the local environment, or from a remote environment via the *Invoke* communication operation. If it is a local dispatcher operation, the coordinator calls *AcquireAndRun* (see Section B.2.2) to start the dispatcher. If it is a remote dispatcher operation, the coordinator sends the token to the remote environment to continue execution of the dispatcher operation there.

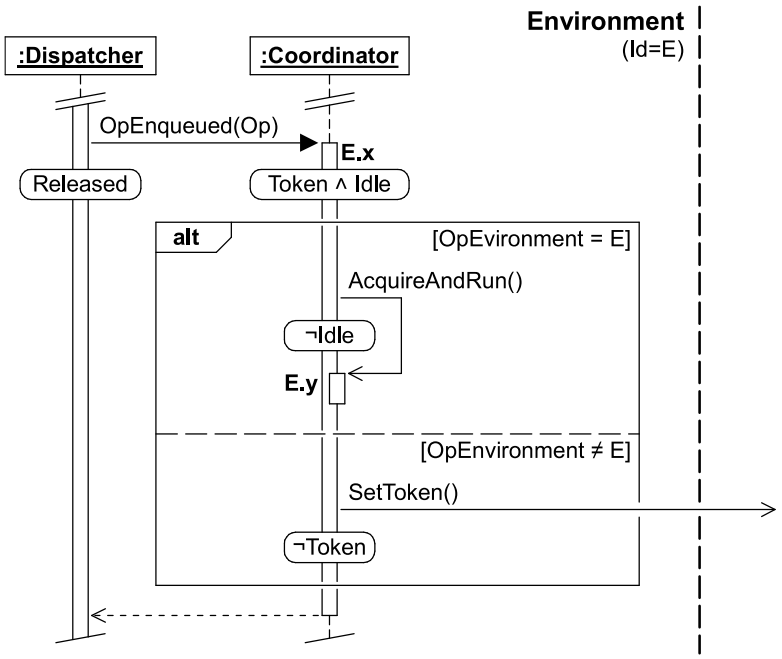


Figure B.14: *OpEnqueued* event handler (*Token ∧ Idle ∧ Released*)

List of Figures

List of Listings

Bibliography

[Abdelnur and Hepper, 2003] Abdelnur, A. and Hepper, S.: *JavaTM Portlet Specification*. Version 1.0, JSR 168. http://download.oracle.com/otndocs/jcp/PORTLET_1.0-FR-SPEC-G-F/, October 2003.

[Anderson, 2006] Andersen, L.: *JDBC™ Specification*. 4.0, JSR-221, Sun Microsystems, Inc., California, USA. http://www.jcp.org/en/jsr/detail?id=221, November 2006.

[Beer, 2000] Beer, W.: *Visuelle Montageumgebung für Java Softwarekomponenten*. Master thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, June 2000.

[Berjon et al., 2013] Berjon, R., Faulkner, S., Leithead, T., Navara, E. D., O'Connor, E., and Pfeiffer, S.: *HTML5. A vocabulary and associated APIs for HTML and XHTML*, World Wide Web Consortium (W3C). http://www.w3.org/TR/html5/, August 2013.

[Berners-Lee et al., 2005] Berners-Lee, T., Fielding, R., and Masinter, L.: *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986, Network Working Group. https://tools.ietf.org/html/rfc3986, January 2005.

[Birman, 2005] Birman, K. P.: *Reliable distributed systems. Technologies, Web services, and Applications*. Springer, New York, 2005. ISBN: 13 978-0-387-21509-9.

[Birrell and Nelson, 1984] Birrell, A.; Nelson, B.: *Implementing remote procedure calls*. In *ACM Transactions on Computer Systems (TOCS)*, Vol. 2. Issue 1, New York, USA, pages 39–59, 1984. doi: 10.1145/2080.357392.

[Birsan, 2005] Birsan, D.: *On plug-ins and extensible architectures*. In *ACM Queue. Patching and Deployment* Volume 3, ACM New York, pages 40–46, 2005. doi: 10.1145/1053331.1053345

[Booth et al., 2004] Booth, D., Haas, H., McCabe F., Newcomer, E., Champion M., Ferris, Ch., and Orchard, D.: *Web Services Architecture*, World Wide Web Consortium (W3C). http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, February 2004.

[Boudreau, 2007] Boudreau, T., Tulach, J., and Wielenga, G.: *Rich client programming. Plugging into the NetBeans platform*. Prentice Hall, Upper Saddle River, NJ, 2007. ISBN: 0132354802.

[Box, 1998] Box, D.: *Essential COM*. Addison Wesley, Reading, Massachusetts, 1998. ISBN: 0201634465.

[Bray et al., 2008] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F.: *Extensible Markup Language (XML) 1.0*. Fifth Edition, World Wide Web Consortium (W3C). http://www.w3.org/TR/xml/, November 2008.

[Bures et al., 2006] Bures, T., Hnetynka, P., and Plasil, F.: *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*. In *Fourth International Conference on Software Engineering Research, Management and Applications*, SERA 2006, IEEE Computer Society, pages 40-48, Seattle, Washington, USA, August 9-11, 2006.

[Bures et al., 2007] Bures, T., Hnetynka, P., Plasil, F., Klesnil, J., Kmoch, O., Kohan, T., and Kotrc, P.: *Runtime Support for Advanced Component Concepts*. In *5th ACIS International Conference on Software Engineering Research, Management & Applications*, SERA 2007, IEEE Computer Society, pages 337–345, Busan, South Korea, August 20-22, 2007.

[Burns, 2013b] Burns, E.: *JavaServer™ Faces Specification*. Version 2.2, JSR-344, Oracle, USA. http://www.jcp.org/en/jsr/detail?id=344, March 2013.

[Cachin et al., 2011] Cachin, C., Guerraoui, R., and Rodrigues, L.: *Introduction to Reliable and Secure Distributed Programming*. Springer, Heidelberg, Dordrecht, London, New York, 2011. ISBN: 978-3-642-15259-7.

[Christensen et al., 2001] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S.: *Web Services Description Language (WSDL)*. Version 1.1, World Wide Web Consortium (W3C). http://www.w3.org/TR/wsdl, March 2001.

[Chumbley et al., 2010] Chumbley, R., Durand, F., Pilz, G., and Rutt, T.: *WS-I Basic Profile Version 2.0*. OASIS, Web Services Interoperability Organization, http://ws-i.org/profiles/basicprofile-2.0-2010-11-09.html, 2010.

[Clement et al., 2004] Clement, L., Hately, A., von Riegen, C., and Rogger, T.: *UDDI Specification*. Version 3.0.2, OASIS. http://www.uddi.org/pubs/uddi_v3.htm, October 2004.

[Crockford, 2006] Crockford, D.: *The application/json Media Type for JavaScript Object Notation (JSON)*, RFC 4627. http://tools.ietf.org/html/rfc4627, July 2006.

[Delisle et al., 2006] Delisle, P., Luehe, J., and Roth, M.: *JavaServer Pages™ Specification*. Version 2.1, JSR-245, Sun Microsystems, Inc., California, USA. http://www.jcp.org/en/jsr/detail?id=245, May 2006.

[DeMichiel and Shannon, 2013] DeMichiel, L. and Shannon, G.: *Java™ Platform, Enterprise Edition (Java EE) Specification*. Version 7, JSR-342, Oracle, USA. http://www.jcp.org/en/jsr/detail?id=342, April 2013.

[Eclipse, 2006] International Business Machines Corp.: *Eclipse Platform Technical Overview*. http://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf, 2006. Accessed April 2012.

[Ecma, 2010] ECMA International: *Common Language Infrastructure (CLI).* Standard ECMA-335, 5th edition. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf, December 2010.

[Ecma, 2011] ECMA International: *ECMAScript Language Specification*. Standard ECMA-262, 5.1 Edition. http://www.ecma-international.org/publications/standards/Ecma-262.htm, June 2011.

[Eder, 2008] Eder, M.: *Content-Watcher: Ein Werkzeug zur Überwachung von Web-Inhalten*. Master thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, 2008.

[Equinox, 2012] The Eclipse Foundation: *equinox OSGi*. http://eclipse.org/equinox/, 2012. Accessed November 2012.

[Fielding et al., 1999] Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T.: *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, Network Working Group. http://tools.ietf.org/html/rfc2616, June 1999.

[Fielding, 2000] Fielding, R.: *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, Unitversity of California, Irvine, http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf, California, Irvine, 2000.

[Freed and Borenstein, 1996] Freed, N. and Borenstein, N.: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046, Network Working Group. http://tools.ietf.org/html/rfc2046, November 1996.

[Garrett, 2010] Garrett, J. J.: *Ajax: A New Approach to Web Applications*. Adaptive Path Inc., http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/, February 2005.

[Gruber, 2010] Gruber, A.: *Konfigurationswerkzeug „Plugin-Explorer" für die Plugin-Plattform Plux.NET*. Bachelor thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, 2010.

[Gudgin et al., 2007] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Karmarkar, A., and Lafon, Y.: *SOAP Version 1.2*, World Wide Web Consortium (W3C). http://www.w3.org/TR/2007/REC-soap12-part1-20070427/, April 2007.

[Haas and Brown, 2004] Haas, H. and Brown, A.: *Web Services Glossary*. W3C Working Group, http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/, February 2004. Accessed November 2013.

[Hagmüller, 2013] Hagmüller, P.: *Plux: Portierung C# nach nach Delphi*. Bachelor thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, 2013.

[Heineman and Councill, 2001] Heineman, G. T. and Councill, W. T.: *Component-based software engineering. Putting the pieces together. Definition of a Software Component and Its Elements*. Addison-Wesley, pages 5-19, Boston, 2001. ISBN: 9780201704853.

[Hepper, 2008] Hepper, S.: *Java™ Portlet Specification*. Version 2.0, JSR-286, IBM Corporation. http://www.jcp.org/en/jsr/detail?id=286, January 2008.

[Hnetynka and Plasil, 2006] Hnetynka, P. and Plasil, F.: *Dynamic Reconfiguration and Access to Services in Hierarchical Component Models*. In *Component-Based Software Engineering. 9th International Symposium*, CBSE 2006. Lecture Notes in Computer Science (LNCS) 4063, Springer Berlin Heidelberg, pages 352–359, Västerås, Sweden, June 29 - July 1, 2006.

[Hribernig, 2012] Hribernig, T.: *Retrofitting Security in Component-based Applications*. Master thesis, Institute for System Software, Johannes Kepler University, Linz, Austria, 2012.

[Jahn, 2009] Jahn, M.: *Entwurf und Implementierung eines Cross-Compilers von Delphi nach C#*. Master thesis, Institute for System Software, Johannes Kepler University, Linz, Austria, 2009.

[Jahn et al., 2010a] Jahn, M., Wolfinger, R., and Mössenböck, H.: *Extending Web Applications with Client and Server Plug-ins*. In *Software Engineering 2010 - Fachtagung des GI-Fachbereichs Softwaretechnik*, SE 2010. LNI 159, GI, pages 33–44, Paderborn, Germany, February 22-26, 2010.

[Jahn et al., 2010b] Jahn, M., Löberbauer, M., Wolfinger, R., and Mössenböck, H.: *Rule-Based Composition Behaviors in Dynamic Plug-In Systems*. In *17th Asia Pacific Software Engineering Conference*, APSEC 2010, IEEE Computer Society, pages 80–89, Sydney, Australia, November 30 - December 3, 2010. doi: 10.1109/APSEC.2010.19

[Jahn et al., 2011] Jahn, M., Wolfinger, R., Löberbauer, M., and Mössenböck, H.: *Composing user-specific web applications from distributed plug-ins*. In *Computer Science - Research and Development*, Springer, pages 1–21, 2011. doi: 10.1007/s00450-011-0182-0

[Kuhrmann, 2004] Kuhrmann, M., Calamé, J. R., and Horn, E.: *Verteilte Systeme mit .NET remoting. Grundlagen - Konzepte - Praxis*. Elsevier, Spektrum, Akad. Verl., München ;, Heidelberg, 2004. ISBN: 3-8274-1545-4.

[Lengauer, 2012] Lengauer, P.: *Trace-based Debugger for Dynamically Composed Applications*. Master thesis, Institute for System Software, Johannes Kepler University, Linz, Austria, 2012.

[Löberbauer et al., 2010] Löberbauer, M., Wolfinger, R., Jahn, M., and Mössenböck, H.: *Testing the composability of plug-and-play components: A method for unit testing of dynamically composed applications*. In *Intelligent Systems and Informatics* (SISY), 2010 8th International Symposium, pages 413–418, Subotica, Serbia, September 10-11, 2010. doi: 10.1109/SISY.2010.5647368

[Löberbauer et al., 2012] Löberbauer, M., Wolfinger, R., Jahn, M., and Mössenböck, H.: *Composition Mechanisms Classified by their Contributor Provision Characteristics*. In *IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*, Subotica, Serbia, September 20-22, 2012.

[Löberbauer, 2012] Löberbauer, M.: *Testing and Debugging of Dynamically Composed Applications*. PhD thesis, Christian Doppler Laboratory for Automated Software Engineering, Institute for System Software, Johannes Kepler University, Linz, Austria, October 2012.

[Makewave, 2013] Makewave: *Knopflerfish. Open Source OSGi Service Platform*, Version 4.0.0. http://www.knopflerfish.org/, 2013.

[McIlroy, 1968] McIlroy, M. D.: *Mass produced software components*. In *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Scientific Affairs Division, NATO, pages 79–87, 7-11 October, 1968.

[Microsoft, 1996] Microsoft News Center: *Microsoft Announces ActiveX Technologies*. http://www.microsoft.com/en-us/news/press/1996/mar96/activxpr.aspx, 1996. Accessed November 2012.

[Microsoft, 1998] Microsoft Developer Network (MSDN): *Introducing Microsoft Transaction Server*. http://msdn.microsoft.com/en-us/library/aa480405.aspx#feedback, 1998. Accessed November 2012.

[Microsoft, 2003] Microsoft TechNet: *RPC Technical Reference*. http://technet.microsoft.com/en-us/library/cc759499.aspx, 2003. Accessed November 2012.

[Microsoft, 2010] Microsoft Developer Network (MSDN): *Managed Extensibility Framework Overview*. http://msdn.microsoft.com/en-us/library/dd460648.aspx, 2010. Accessed April 2012.

[Microsoft, 2011a] Microsoft Developer Network (MSDN): *.NET Framework Remoting Overview*. http://msdn.microsoft.com/en-us/library/vstudio/kwdt6w2k(v=vs.100).aspx, 2011.

[Microsoft, 2011b] Microsoft: *Microsoft Silverlight 5*. http://www.microsoft.com/silverlight, 2011. Accessed April 2012.

[Microsoft, 2012a] Microsoft Developer Network (MSDN): *About Dynamic Data Exchange (Windows).* http://msdn.microsoft.com/en-us/library/windows/desktop/ms648774(v=vs.85).aspx, 2012. Accessed February 2013.

[Microsoft, 2012b] Microsoft Developer Network (MSDN): *OLE Background*. http://msdn.microsoft.com/en-us/library/19z074ky.aspx, Accessed November 2012.

[Microsoft, 2012c] Microsoft Developer Network (MSDN): *The Component Object Model (COM)*. http://msdn.microsoft.com/en-us/library/windows/desktop/ms694363(v=vs.85).aspx, 2012. Accessed November 2012.

[Microsoft, 2012d] Microsoft TechNet: *DCOM Technical Overview*. http://technet.microsoft.com/en-us/library/cc722925.aspx, Accessed November 2012.

[Microsoft, 2012e] Microsoft Developer Network (MSDN): *COM+ (Component Services)*. http://msdn.microsoft.com/en-us/library/windows/desktop/ms685978(v=vs.85).aspx, 2012. Accessed November 2012.

[Microsoft, 2012f] Microsoft Developer Network (MSDN): *Message Queuing (MSMQ)*. http://msdn.microsoft.com/en-us/library/ms711472.aspx. Accessed November 2012.

[Microsoft, 2012g] Microsoft Developer Network (MSDN): *Overview of the.NET Framework*. http://msdn.microsoft.com/en-us/library/zw4w595w.aspx. Accessed November 2012.

[Microsoft, 2012h] Microsoft Developer Network (MSDN): Visual Studio 2012. http://msdn.microsoft.com/en-us/library/dd831853.aspx. Accessed November 2012.

[Microsoft, 2012i] Microsoft Developer Network (MSDN): *Windows Communication Foundation (WCF)*. http://msdn.microsoft.com/en-us/library/dd456779.aspx, 2012. Accessed November 2012.

[Microsoft, 2012j] Microsoft Developer Network (MSDN): *Active Server Pages*. http://msdn.microsoft.com/en-us/library/aa286483.aspx. Accessed November 2012.

[Microsoft, 2012k] Microsoft: *ASP.NET. Web Development with Power, Productivity & Speed*. http://www.asp.net/, 2013. Accessed August 2013.

[Microsoft, 2012l] Microsoft Developer Network (MSDN): *Common Language Runtime (CLR)*. http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx. Accessed November 2012.

[Microsoft, 2013a] Microsoft Developer Network (MSDN): *Named Pipes*. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365590(v=vs.85).aspx, 2013.

[Microsoft, 2013b] Microsoft Developer Network (MSDN): *Application Domains*. http://msdn.microsoft.com/en-us/library/2bh4z9hs.aspx. Accessed August 2013.

[Microsoft, 2013c] Microsoft: *Internet Information Services (IIS)*. http://www.iis.net/, 2013. Accessed August 2013.

[Mittermair, 2010] Mittermair, C.: *Zerlegung eines monolithischen Softwaresystems in ein Plug-In-basiertes Komponentensystem*. Master thesis, Institute for System Software, Johannes Kepler University, Linz, Austria, 2010.

[Mordani, 2009] Mordani, R.: *Java™ Servlet Specification*. Version 3.0, Sun Microsystems, Inc., JSR-315. http://www.jcp.org/en/jsr/detail?id=315, December 2009.

[Muskalla and Sternberg, 2007] Muskalla, B. and Sternberg, R.: *RCP goes Web 2.0. Web-enabled RCP Applications with the Rich Ajax Platform*. In *Eclipse-Magazin*, Vol. 12, 2007.

[NPAPI, 2012] MozillaWiki: *NPAPI Specifications*. https://wiki.mozilla.org/NPAPI#Specifications, September 2012.

[OASIS, 2013] OASIS (Organization for the Advancement of Structured Information Standards): *Web Services Interoperability (WS-I)*. http://www.oasis-ws-i.org/. Accessed August 2013.

[Oliphant, 1996] Oliphant, Z.: *Programming Netscape plug-ins*, Sams Net, Indianapolis, 1996. ISBN: 1575210983.

[OMG, 2006] Object Management Group (OMG): *Meta Object Facility (MOF) Core Specification*. Version 2.0, http://www.omg.org/spec/MOF/2.0/PDF/, January 2006.

[OMG, 2011] Object Management Group (OMG): *Unified Modeling Language (UML)*, Specification, Version 2.4.1. http://www.omg.org/spec/UML/2.4.1/, August 2011.

[OMG, 2012] Object Management Group, Inc. (OMG): *Common Object Request Broker Architecture (CORBA)*, Specification, Version 3.3. http://www.omg.org/spec/CORBA/3.3/, November 2012.

[Oracle, 2010] Oracle: *Java RMI Specification*, Java SE 7. http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html, 2010.

[Oracle, 2013a] Oracle: *Java Development Kit (JDK)*. http://www.oracle.com/technetwork/java/index.html, October 2013.

[Oracle, 2013b] Oracle: *The NetBeans Platform*, https://netbeans.org/features/platform/index.html. Accessed November 2013.

[Oracle, 2013c] Oracle: *A Brief History of NetBeans*, https://netbeans.org/about/history.html. Accessed December 2013.

[Oracle, 2013d] Oracle: *Happy Birthday NetBeans. Interview with Jaroslav "Yarda" Tulach*. https://netbeans.org/community/articles/interviews/yarda-tulach.html. Accessed December 2013.

[Oracle, 2013e] Oracle: *Lesson: Java Applets*, http://docs.oracle.com/javase/tutorial/deployment/applet/. Accessed December 2013..

[OSGi Alliance, 2012a] OSGi™ Alliance: *OSGi Release 5*. http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf, March 2012. Accessed November 2013.

[OSGi Alliance, 2012b] OSGi™ Alliance: *OSGi Enterprise Release 5*. www.osgi.org/download/r5/osgi.enterprise-5.0.0.pdf, March 2012. Accessed November 2013.

[Rammer, 2005] Rammer, I. and Szpuszta, M.: *Advanced .NET remoting*. Apress; Distributed by Springer-Verlag, Berkeley, CA, New York, 2005. ISBN: 1-59059-417-7.

[RAP, 2012] *RAP - Rich Ajax Platform*. *Enabling modular business apps for desktop, browser and mobile*. http://www.eclipse.org/rap/, 2012. Accessed November 2012.

[Reinthaler, 2012] Reinthaler, T.: *Deployment Assistant for Plux*. Master thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, 2012.

[Reiter and Wolfinger, 2007] Reiter, S. and Wolfinger, R.: *Erfahrungen bei der Portierung von Delphi Legacy Code nach .NET*. In *Software Engineering 2007, Fachtagung des GI-Fachbereichs Softwaretechnik*, SE 2007. LNI 105, GI, pages 353–356, Hamburg, Germany, March 27-30, 2007.

[Robinson and Coar, 2004] Robinson, D. and Coar, K.: *The Common Gateway Interface (CGI) Version 1.1*. RFC 3875, Network Working Group. http://tools.ietf.org/html/rfc3875, October 2004.

[Pichler, 2009] Pichler, R.: *Metrix - A Measuring Tool for Run-time Figures in Plug-in based .NET Applications*. Bachelor thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, 2009.

[Postel, 1980] Postel, J.: *User Datagram Protocol (UDP)*. RFC 768. http://tools.ietf.org/html/rfc768, August 1980.

[Postel, 1981] Postel, J.: *Transmission Control Protocol - DARPA Internet Program Protocol Specification*. RFC 793, Defense Advanced Research Projects Agency, Information Processing Techniques Office, Virginia. http://tools.ietf.org/html/rfc793, September 1981.

[Schell, 1971] Schell, R. R.: *DYNAMIC RECONFIGURATION in a MODULAR COMPUTER SYSTEM*. Technical Report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1971.

[Schenkermayr, 2013] Schenkermayr, B.: *Ein komponentenbasierter Taschenrechner auf Basis von Plux*. Master thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, 2013.

[Spasov, 2013] Spasov, N.: *Porting the Plugin Platform Plux to Java*. Master thesis, Institute for System Software, Johannes Kepler University, Linz, Austria, 2013.

[Srinivasan, 1995] Srinivasan, R.: RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Network Working Group. http://tools.ietf.org/html/rfc1831, August 1995.

[Sun Microsystems, 1996] Sun Microsystems Inc.: *JavaBeans (TM) API specification*, 2550 Garcia Avenue, Mountain View, CA 94043, Version 1.00-A. ftp://sunsite.univie.ac.at/pub/languages/java/splash.javasoft.com/pub/beans.100A.pdf, December 1996.

[Szyperski, 2002] Szyperski, C., Gruntz, D., and Murer, S.: *Component software. Beyond object-oriented programming*. ACM Press; Addison-Wesley, New York, London, Boston, 2002. ISBN: 0201745720.

[Tanenbaum and Van Steen, 2007] Tanenbaum, A. and Van Steen, M.: *Distributed systems. Principles and Paradigms*. Pearson Prentice Hall, Upper Saddle River, NJ, 2007. ISBN: 0-13-239227-5.

[The Apache Software Foundation, 2008] The Apache Software Foundation: *Apache Felix*. http://felix.apache.org/, 2008.

[The Apache Software Foundation, 2013] The Apache Software Foundation: *Apache httpd Tutorial: Introduction to Server Side Includes*. http://httpd.apache.org/docs/current/howto/ssi.html, November 2013.

[The Open Group, 1999] The Open Group: *The ActiveX Core Technology Reference*, Reading, Berkshire, 1999.

[The PHP Group, 2012] The PHP Group: *PHP: Hypertext Preprocessor*. http://www.php.net/manual/en/preface.php, 2013. Accessed November 2012.

[Thurlow, 2009] Thurlow, R.: *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC 5531, Network Working Group. http://tools.ietf.org/html/rfc5531, May 2009.

[Vatkina, 2013] Vatkina, M.: *Enterprise JavaBeans™. EJB Core Contracts and Requirements*. Version 3.2, JSR-345, Oracle, USA. http://www.jcp.org/en/jsr/detail?id=345, April 2013.

[Vogel, 2009] Vogel, O.: Software-Architektur. Grundlagen - Konzepte - Praxis. Spektrum, Akad. Verl., Heidelberg, 2009. ISBN: 978-3-8274-1933-0.

[Weinreich and Sametinger, 2001] Weinreich, R. and Sametinger, J.: *Component-based software engineering*. *Putting the pieces together*. *Component Models and Component Services: Concepts and Principles*. Addison-Wesley, pages 33-48, Boston, 2001. ISBN: 9780201704853.

[Weiss, 2010] Weiss, S.: *Beispielprogramm Kundenbeziehungsmanagement "Plux-CRM" für die Plugin-Plattform Plux.NET*. Master thesis, Institute for Systemsoftware, Johannes Kepler University, Linz, Austria, 2010.

[Winer, 1999] Winer, D.: *XML-RPC Specification*, UserLand Software. http://xmlrpc.scripting.com/spec.html, June 1999.

[Winer, 2003] Winer, D.: *RSS 2.0 Specification*, Berkman Center for Internet & Society at Harvard Law School. http://cyber.law.harvard.edu/rss/rss.html, July 2003.

[White, 1976] White, J.: *A High-Level Framework for Network-Based Resource Sharing*, Stanford Research Institute, California, RFC 707. http://tools.ietf.org/html/rfc707, January 1976.

[Wolfinger et al., 2006] Wolfinger, R., Dhungana, D., Prähofer, H., and Mössenböck, H.: *A Component Plug-In Architecture for the .NET Platform*. In *Modular Programming Languages, 7th Joint Modular Languages Conference*, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings. Lecture Notes in Computer Science 4228, Springer, 287–305, 2006. doi: 10.1007/11860990_18.

[Wolfinger and Prähofer, 2007] Wolfinger, R. and Prähofer, H.: *Integration models in a .NET plug-in framework*. In *Software Engineering 2007, Fachtagung des GI-Fachbereichs Softwaretechnik*, SE 2007. LNI 105, GI, pages 217–230, Hamburg, Germany, March 27-30, 2007.

[Wolfinger et al., 2008] Wolfinger, R., Reiter, S., Dhungana, D., Grünbacher, P., and Prahofer, H.: *Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques*. In *7th IEEE International Conference on Composition-Based Software Systems*, ICCBSS 2008, IEEE Computer Society Press, pages 21–30, Madrid, Spain, February 25-29, 2008. doi:10.1109/ICCBSS.2008.30

[Wolfinger, 2010] Wolfinger, R.: *Dynamic Application Composition with Plux.NET. Composition Model, Composition Infrastructure*. PhD thesis, Christian Doppler Laboratory for Automated Software Engineering, Institute for System Software, Johannes Kepler University, Linz, Austria, January 2010.

[Wolfinger et al., 2010] Wolfinger, R., Löberbauer, M., Jahn, M., and Mössenböck, H.: *Adding genericity to a plug-in framework*. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE 2010, ACM; Association for Computing Machinery, pages 93–102, Eindhoven, The Netherlands, October 10-13, 2010. doi: 10.1145/1868294.1868308.

[Wolfinger et al., 2012] Wolfinger, R., Löberbauer, M., Jahn, M., and Mössenböck, H.: *Retrofitting Security in Component-based Applications*. In *Computer Science - Research and Development*. Springer, 2012 (submitted for publication).

Curriculum Vitae

Name:	Markus Jahn
Date of birth:	March 6, 1982
Place of birth:	Freistadt, Austria
Nationality:	Austria
Contact:	markus.jahn@jku.at

Education

2009-2014	Doctorate Degree in Technical Sciences Johannes Kepler University, Linz
2007-2009	Master's Degree in Computer Science Johannes Kepler University, Linz Graduated with distinction
2006-2007	Study abroad in Computer Science Dublin City University, Ireland
2002-2007	Bachelor's Degree in Computer Science Johannes Kepler University, Linz
1996-2001	Higher technical school for Building Construction, Linz Graduated with distinction
1992-1996	Secondary school, Grünbach
1988-1992	Primary school, Grünbach

Professional Career

2013-	Software Engineer, Wolfinger Software, Linz
2008-2013	Research Assistant, Christian Doppler Laboratory for Automated Software Engineering Johannes Kepler University, Linz