JOHANNES KEPLER
UNIVERSITY LINZ | JKU

Dynamic Application Composition with Plux.NET

Composition Model, Composition Infrastructure

Dissertation
Submitted in fulfillment of the requirements for the academic degree

Dr. rer. soc. oec.

Doctor of Social and Economic Sciences

Completed at
Christian Doppler Laboratory for Automated Software Engineering
Institute for System Software

By
Reinhard Wolfinger

Supervised by
o. Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck
a. Univ.-Prof. Dr. Johannes Sametinger

Linz, *January 2010*

A-4040 Linz, Altenberger Straße 69, Internet: http://www.jku.at, DVR 0093696

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 verwendet und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen deutlich als solche kenntlich gemacht habe.

Linz, im Jänner 2010 Reinhard Wolfinger

Abstract

Although modern applications are often designed to have a component-based architecture, they are usually deployed as a monolithic piece. The monolith causes problems when applications get feature-rich and should be made customizable. *Dynamic composition* allows developers to build an application where users only load components they need for their current work. This keeps the application small and simple. Moreover, dynamic composition means that an application can be reconfigured on the fly by dynamically swapping sets of components without programming or configuration.

Plux.NET is a *component model and infrastructure* for dynamic composition: The component model specifies requirements and provisions among components declaratively using the component's metadata. The discovery core supports automatic discovery of components using exchangeable discovery mechanisms. The composition core uses the metadata to compose an application by matching requirements and provisions, and stores connections between components in the composition model. Component developers use an event-based programming model, which gives host components a uniform mechanism to integrate contributor components at startup as well as at run time when an application dynamically changes.

If an application can be reconfigured while it is running, then the user interface must also change dynamically. Best practice guidelines for user interface design show how to consider that the user interface will be adaptable at run time. Special widgets bind to the composition model and simplify the implementation of dynamic user interfaces, because they automatically update their content and state when the composition model changes.

Kurzfassung

Moderne Programme haben eine komponentenbasierte Architektur und werden dennoch meist als Monolith ausgeliefert. Das verursacht Probleme, wenn man große Programme mit vielen Funktionen für einzelne Benutzer anpassen will. Mit dynamischer Komposition lassen sich Programme entwerfen, bei denen ein Benutzer nur jene Komponenten lädt, die er für seine aktuelle Aufgabe benötigt. Das hält Programme klein und einfach. Wenn sich die Aufgabe des Benutzers ändert, konfiguriert er das Programm um, während es läuft. Dabei tauscht er die nicht mehr benötigten Komponenten durch andere Komponenten aus. Dieser Vorgang erfordert weder Programmierung noch Konfiguration.

Plux.NET ist ein Komponentenmodell und eine Infrastruktur für dynamische Komposition. Das Komponentenmodell spezifiziert Anforderungen und Garantien zwischen Komponenten deklarativ mit Steckplätzen und Steckern in den Metadaten der Komponenten. Der *Discovery*-Kern unterstützt dynamisches Entdecken von Komponenten mittels austauschbarer Mechanismen. Der *Composer*-Kern baut eine Anwendung auf Basis der Metadaten zusammen, indem er Stecker und Steckplätze verbindet. Entwickler verwenden ein ereignisbasiertes Programmiermodell. Der Mechanismus zur Integration anderer Komponenten ist dabei einheitlich, egal ob Komponenten beim Programmstart, oder zur Laufzeit, wenn das Programm angepasst wird, zusammengesteckt werden.

Wenn Programme angepasst werden während sie laufen, muss sich die Benutzerschnittstelle dynamisch ändern. *Best Practice*-Richtlinien beschreiben wie man eine Benutzerschnittstelle so entwirft, dass sie zur Laufzeit angepasst werden kann. An Steckplätze gebundene Steuerelemente vereinfachen die Implementierung von dynamischen Benutzerschnittstellen, weil sie ihren Inhalt und Zustand bei Änderungen im Kompositionsmodell automatisch ändern.

Table of Contents

Chapter 1: Introduction

Modern applications are often designed to have a component-based architecture and are built with object-oriented application frameworks. In a nutshell, a component is a modular part of a system. It defines its behavior in terms of provided and required interfaces. The process to construct an application from components is essentially a matching of requirements and provisions. This component assembly process is called composition.

In practice, decomposition produces rather coarse-grained business logic or presentation components and features are usually deployed as a monolithic piece. Existing composition approaches perform component assembly rather early in an application's life-cycle. Static approaches compose programmatically and set everything in stone already at compile time. Partly dynamic approaches compose at startup and actually allow coarse-grained dynamic additions, but fail when components should be dynamically removed. This causes problems when applications get feature-rich and should be made customizable.

This thesis presents a fully dynamic approach to composition. The approach follows the plug and play metaphor allowing composition of applications without programming. Dynamic composition allows developers to build applications where users only load those components which they need for their current work. Moreover, dynamic composition means that an application can be reconfigured on the fly by dynamically swapping sets of components. This keeps applications small, simple, and always aligned with the working situation at hand.

Section 1.1 discusses why customization, extensibility and dynamic reconfiguration are important in the business software domain. Section 1.2 explains why dynamic composition is useful for any feature-rich application and what problems current component frameworks have in regard to dynamic composition. Section 1.3 explains the research process and highlights the contributions of this thesis. Section 1.4 overviews the history of the Plux.NET project, introduces the Plux.NET team, and lists results of other Plux.NET contributors. Section 1.5 outlines the remaining chapters of this thesis.

1.1 Research Context

The initiative to this thesis came from the business software industry. Our research project was conducted in close collaboration with an industry partner: BMD Systemhaus GmbH is a medium-sized company offering Enterprise Resource Planning (ERP) software mainly to

small and medium enterprises. The company has a significant market share in Austria, Germany, and Hungary.

Rich client business software typically has a component-based architecture and is deployed in coarse-grained components. Eventually, all customers get the same application, while configuration or license codes determine whether particular features are enabled or not. This one-size-fits-all approach causes three major problems for the manufacturer of the business application:

- Business software is inherently complex and feature-rich, while individual users only need a small fraction of the features. Hence, if the user interface of an application is cramped with features for all business processes, users struggle to find features they really need for their tasks.
- Customer requirements for business applications are characterized by a large variation and depend on industry or company size. It is impossible for a business application of any size to fully meet customer requirements with an off-the-shelf product. Even if an application covers the major business-relevant scenarios, customers typically ask for more features addressing their special needs. Some of these features are highly industry-specific and thus outside the manufacturer's core competence. Thus the manufacturer may decide to not include them in the product. Still, customers need a way to add features that are important to them.
- Business customers tend to be conservative about deploying patches. Often one business unit urges to deploy a certain patch, while another business unit is reluctant to do so yet. The coarse-grained deployment model assumes that patching means to replace large parts of the application. This rules out selective patch deployment scenarios.

BMD Systemhaus derived the following goals for a new generation of business software from these problems. Those goals should be generally worthwhile for any large application:

- The business application should be made *customizable* to the needs of individual users. The application should be broken up into a slim core application that can be extended with features tailored to the user's needs. When a user changes to a new working situation the application should adapt. The application *dynamically adds* features for the new working situation, and it *dynamically removes* features which are no longer needed.
- The business application should me made *extensible* in order to close the gap between what features customers need and what the manufacturer can provide in the base product. End users and third parties should be able to contribute any functionality the manufacturer did not already provide in the base product.

In cooperation with BMD we have developed a set of usage scenarios demonstrating the need for a reconfigurable application with support for dynamic addition and removal. The scenarios are motivated by the ERP application domain and BMD's market environment.

Scenario 1: Role-specific views. Business software is inherently feature-rich as large enterprises need to support a high number of success-critical business processes. Individual users often participate only in a few of these processes. Hence, the user interface of an application is often cramped with features not needed for a particular task. Dynamic reconfiguration en-

ables customization of user interfaces to individual tasks and responsibilities on the fly. This helps improving focus and reducing clutter. Users are involved in diverse business processes and tasks during their working day. Dynamic reconfiguration relies on feature configurations for the different roles and dynamic switching of roles.

Scenario 2: Optimizing training. A major problem in training is that new users are often overwhelmed by the high number of features of the software application. A trainer explaining a basic feature has to guide the trainees through numerous menus and user dialogs to activate a function needed for the next training. Obviously, it is more promising to offer an individually configured system to trainees in accordance with the training schedule. This allows starting with a small configuration showing only some basic functions and adding new features for each new training unit. Training can thus be organized in small steps adding complexity incrementally and in coordination with the training program.

Scenario 3: On-the-fly product customization for sales process. Sales staff offers products based on pre-defined feature sets to customers. Usually, this sales process leads to long lasting discussions with customers about the value and cost of features as customers cannot explore and experience the system before it is purchased and installed. Dynamic reconfiguration supports a more rapid and interactive sales process. Salespersons explore feature combinations together with customers, and rapid reconfiguration of the application allows a live preview of the system by the customer taking into account the IKIWISI ("I know it when I see it") phenomenon. The salesperson can instantly demonstrate the software in the desired configuration and explain the provided functions.

Scenario 4: Renting features. It is an interesting business case for customers to rent and use particular product features for a limited period instead of purchasing them permanently. Dynamic reconfiguration allows customers browsing the available rentable features, immediately installing them from a remote site, trying out features during an evaluation period, and using the features for a defined period. Customers can continuously keep track of the accumulated rental fees.

A business application that is customizable, extensible, and does support these dynamic reconfiguration scenarios requires a composition approach with support for dynamic addition and removal. The next section discusses why existing composition approaches and application frameworks do not adequately support dynamic reconfiguration.

1.2 Problem Statement

This thesis pursues the problem of how to build customizable, extensible, and reconfigurable applications. Section 1.1 discussed why these characteristics are important for ERP applications. These were just motivating examples. Extensibility, customization and dynamic reconfiguration are universally applicable non-functional requirements for any domain, as soon as an application grows larger. How such applications can be built, and what an adequate architecture looks like, is an open research problem.

If the problem can be solved, developers could build applications where users will load only components they need for their current work thus keeping the application small and simple. Applications can be reconfigured on the fly for different user roles by dynamically swapping

sets of plug-in components. Applications seamlessly integrate extensions from third-parties, where the manufacturer did not provide the functionality in the base product.

The starting point to solve this problem is too look at existing component technologies and question where they fail. The reason why most existing composition approaches do not adequately support dynamic reconfiguration is that they perform component assembly too early in the process. Composition approaches can be grouped in three categories. Firstly, approaches which assemble components at compile time, such as hard-wired Java programs, cannot be reconfigured at all. Secondly, approaches which assembly components at startup time, such as Dependency Injection Frameworks (Pico 2009), or Service Loaders (Sun 2006, Boudreau et al. 2007) can be reconfigured, but require a restart. Thirdly, the dynamic approaches in current Plug-in Frameworks (Eclipse 2003, OSGi 2006, Chatley et al. 2004) rudimentarily allow users to dynamically add components without requiring a restart, but they completely fail when components should be dynamically removed.

Composition approaches that require component assembly to happen at development time are completely inappropriate for our purpose. Other approaches that rely on programmatic effort for assembling components (Eclipse 2003, OSGi 2006) are not automated enough. Recently the concept of plug-in components emerged as a promising way of building applications which are inherently extensible and customizable. The plug-in approach is based on the concept of a small core application which is extended with plug-in components. Plug-ins can plug into the core application or into other plug-ins where they are integrated seamlessly by their host.

Several plug-in systems have already found their way into software development practice (Eclipse 2003, OSGi 2006). Despite the success of plug-in systems so far, they still suffer from several deficiencies. Plug-in systems provide some of the mechanisms required for reconfigurable applications, but some open issues remain:

- *Lack of granularity.* A plug-in application depends on a set of deployed components. The deployed configuration may vary per user, which allows customized applications. Current plug-in frameworks support only one granularity level. Often the smallest unit of variability is a plug-in, which is a rather coarse-grained customization granularity. Although, the underlying component model could be misapplied by designing for numerous small plug-ins, the composition operations would still support only the one granularity level. With only one granularity level the architect has two choices: Either he designs for coarse-grained plug-ins and his customization options are limited. Or he designs for fine-grained plug-ins and the lacking structure makes reconfiguration confusing and cumbersome.

  Another aspect of granularity is the question which parts of the system are affected by a change. In existing plug-in frameworks changes always have system-wide effect. It is not possible to add or remove components only in selective parts of the application. The open research question is: *how can we provide more fine-grained customization?*

- *Plug-in integration requires programmatic effort.* In plug-in systems, such as OSGi or Eclipse, developers face significant programming effort when they deal with plug-in integration. In the extensibility model of plug-in systems the plug-in host is the active part. The programmer of the host has to provide code that looks up plug-ins in a reg-

istry, code that instantiates the plug-in, and more code that connects the plug-in host with the plug-in. The open research question is: *how can we automate the integration of plug-ins?*

- *Dynamic change support is optional.* The composition approach in most plug-in systems is focused on wiring components at startup, and provides basic support for dynamic addition. But the programming model neither requires plug-in hosts to be aware of plug-ins being removed, nor do the programming interfaces provide operations to remove components. The open research question is: *how does a programming model for dynamic reconfiguration look like?*
- *Non-uniform programming model for startup and dynamic change.* The plug-in programming model is not uniform, in the sense that the code for integrating a plug-in at startup is different from the code for integrating it at run time. The programmer of the host plug-in has to provide two different implementations. The open research question is: *how does a uniform programming model for plug-in integration look like?*

The lack of dynamic reconfiguration in current plug-in systems can be observed in the market place. Plug-in based application are indeed customizable and extensible, and they can be reconfigured, but many need to be restarted when a plug-in is added, and all of them need to be restarted when a plug-in is removed. Since this thesis envisions applications that can be reconfigured on the fly without being restarted, the issues need to be addressed.

1.3 Research Approach and Contributions

The research method in a scientific discipline depends on which research goal the science follows. This is a thesis in software engineering, the area of research is component-based architecture. The goal is to create a composition model and a prototypical implementation of a composition infrastructure.

The used research methodology follows the experimental paradigm. The experimental paradigm requires an experimental design, observation, data collection and validation on the artifacts being studied. According to the classification of Basili, this thesis uses the engineering method, which is a variation of the scientific method (Basili 1993). The engineering method is applied in this thesis with the following research process:

a) *Observe existing solutions*. Systematically analyze existing component models on the basis of sample applications to study their characteristics in terms of dynamic reconfiguration. In particular with regard to the research issues developed in the problem statement (see page 3-5).

b) *Propose better solution*. Design a composition model based on existing models with improved support for dynamic reconfiguration. The particular focus is on fine-grained customizability, dynamic addition and removal, automated integration, and a simple uniform programming model.

c) *Build/develop.* Implement the composition model in a composition framework. The prototypical implementation is the technical foundation for the case studies. The

framework implementation allows building applications that demonstrate the capabilities of the composition model and its practical applicability.

d) *Measure and analyze.* Measure and analyze characteristics in dynamic reconfiguration on the basis of the case studies.

e) *Repeat the process until no more improvements are possible.* Repeat steps 1 to 4 until deficiencies of existing component models are clarified. Then repeat steps 2 to 4 until the case studies show the aspired improvements in measurements.

The approach is oriented on evolutionary improvement. It analyzes existing component models, and modifies or refines aspects of the model in order to improve the aspects being studied.

The defined research process produced the following contributions:

- *Composition Model*. The Plux.NET composition model enables plug-and-play composition of plug-in-based applications. The three novelties of the composition model are: First, unlike existing plug-in systems, the composition model does not operate as a passive registry where the components themselves drive composition. Instead, the composition model defines a composition service which actively controls composition. Second, the composition model stores which host component uses which contributor component. Thirdly, the components adhere to an event-based programming model. They react to the event notifications of the composition service. These three novelties are the key to applications which can be extended and reconfigured without a restart. The composition service plugs together a set of components for one working context, and swaps them with another set of components for another working context.

  Section 3 presents the Plux.NET composition model.

- *Composition Framework.* The Plux.NET composition framework illustrates how to implement the composition model in a programming model that automates plug-in integration, and uniformly handles startup configuration and dynamic reconfiguration. The composition framework refines the concept of a plug-in framework with a focus on dynamic reconfiguration. The novelties are fine-grained customization support, declarative composition with automated plug-in integration, and a uniform programming model for composition at startup, or configuration changes while the application is running.

  Section 4 presents the Plux.NET composition infrastructure. Section 5 gives an introduction on how to build applications with the composition framework.

1.4 Project History

The Plux.NET project is carried out at the Christian Doppler Laboratory for Automated Software Engineering associated with the Institute for System Software at the Johannes Kepler University Linz. At the time of this writing the Plux.NET team members are Reinhard Wolfinger, Markus Jahn, and Markus Löberbauer.

The research project has been conducted in close collaboration with our industry partner BMD Systemhaus GmbH. BMD's software product is a comprehensive suite of ERP applications for customer relationship management, accounting, cost accounting, payroll, enterprise resource planning, as well as production planning and control. BMD's target market is fairly diversified, ranging from small tax counselors to medium-sized auditing firms or large corporations. Customized products are an essential part of BMD's marketing strategy to address the needs of those markets. The results of this thesis should create momentum for a customizable next generation of BMD software.

First ideas of Plux.NET go back to the master thesis of Deepak Dhungana in 2006 (Dhungana 2006). He developed a plug-in framework for Microsoft .NET called Client Application Platform .NET (CAP.NET). CAP.NET realized extension points known from the Eclipse Platform for Microsoft's .NET platform. Dhungana changed the mechanism to specify extension points. Where Eclipse uses XML files to specify provided and required interfaces, CAP.NET uses .NET attributes to declare the composition aspects directly in the source code of an application. The benefit is that CAP.NET specifies everything in one place, where Eclipse needs two separate artifacts. Building on CAP.NET, we adapted the terminology of CAP.NET to introduce Plux.NET's slot and plug metaphor and published the extensibility model (Wolfinger 2006).

In early 2007, we published a first generation prototype of the Plux.NET composition framework. With the composition runtime built by Stephan Reiter, we demonstrated for the first time how we can compose an application in a plug and play fashion by adding and removing components while the application is running. Our visualization tool instantly showed the architectural changes. Together with Herbert Prähofer, we published models for host and plug-in integration that addressed execution of plug-ins in reliable settings and allowing independent evolution of core applications and plug-ins (Wolfinger 2007).

Stephan Reiter has worked on a case study where he ported the customer relationship management (CRM) application of our industry partner from Borland Delphi to Delphi.NET. The port was preparatory work for an initiative to decompose the monolithic CRM application into plug-ins. We published an experience paper with our porting experiences (Reiter 2007).

Later in 2007, we started a cooperation with the product line engineering group from our laboratory. Together with Stephan Reiter, Deepak Dhungana, Paul Grünbacher, and Herbert Prähofer, we designed an approach to integrate a component framework with a product line engineering tool suite. During this cooperation we also developed the new usage scenarios described in the research context section of this chapter (see pages 1-3). We published the results of the cooperation in a paper (Wolfinger 2008a).

In summer 2007, Christian Mittermair joined the team to decompose and re-architecture the CRM case study that Stephan Reiter ported to .NET as a plug-in-based application. He completed the project in spring 2009 and also contributed to the Plux.NET framework.

In early 2008, Markus Jahn joined the Plux.NET project and built the second generation prototype of the composition framework. Shortly after that, the first Plux.NET-based application was completed. Mario Eder built *ContentWatcher* for his master thesis, a tool that crawls the web to detect updated content (Eder 2008). Sabine Weiss started a master thesis project with the goal to create a Plux.NET reference application. The reference application emphasizes

fine-grained decomposition and fully utilizes Plux.NET to create a highly customizable and extensible application. In summer 2008, we narrowed our research issues and published the intended research process at the OOPSLA 2008 Doctoral Symposium (Wolfinger 2008b).

Early in 2009, we continued our cooperation with the product line group from our laboratory. Rick Rabiser used our CRM case study to illustrate his three-level customization approach with software product lines. We published the approach and the results of the case study (Rabiser 2009). Markus Jahn completed the yet most comprehensive Plux.NET-based application. The cross-compiler *Atac* and the corresponding integrated development environment (IDE) showed how to build applications with a composition-oriented programming style (Jahn 2009a).

In 2009, Markus Löberbauer joined Plux.NET as an architect and we created the third-generation prototype with a focus on stability. We cleaned up the programming interfaces and the implementation. The development tools group of our industry partner lead by Horst Hagmüller initiated a project to build a pilot version of their next-generation ERP applications based on Plux.NET. Markus Jahn has started his PhD project where he has brought the idea of dynamic composition and run-time reconfiguration to web applications (Jahn 2009b).

Also in 2009, Andreas Gruber finished the implementation of his bachelor project where he developed a graphical composition tool for Plux.NET. Zóltan Tóth, Rainer Pichler, and Mario Mlinaric implemented a script interpreter based on Microsoft Powershell that allows configuring Plux.NET applications with a scripting language. Rainer Pichler completed his bachelor project Metrix, which is a measurement tool for the Plux.NET run-time environment. He published his bachelor thesis in autumn 2009 (Pichler 2009).

1.5 Structure of the Thesis

This thesis is organized as follows: Chapter 2 discusses the state of the art in dynamic composition. A historical overview shows how component-based software engineering evolved from the beginnings of modularization to plug-in frameworks. A definition section introduces important terminology for component technology. Then a section introduces typical representatives of current component systems, before a detailed analysis explains the deficiencies of existing component systems.

Chapter 3 describes a composition model which addresses the deficiencies described in Chapter 2. The composition model defines meta elements for components and their relationships, and it specifies services for discovery, qualification, and composition. The composition model is the foundation for the composition infrastructure described in Chapter 4.

Chapter 4 describes a composition infrastructure which implements the composition model described in Chapter 3. The composition infrastructure allows building rich client applications which support fine-grained customization and dynamic reconfiguration using plug-and-play composition.

Chapter 5 describes how to design and implement applications with the Plux.NET application programming interface (API). The Plux.NET runtime core is universally applicable and can be used for any kind of .NET application.

Chapter 6 summarizes the main contributions, recapitulates how those contributions address the problem statement, and concludes the thesis with an outlook on future research.

Chapter 2: State of the Art

Assembling systems from pre-fabricated building blocks has been regarded as an appealing approach to software construction since the early days of software engineering. Although we have seen much progress into this direction in the last decades, component-based software engineering still has not reached a level of maturity that is comparable to other engineering disciplines. Building software systems by assembling components in a plug-and-play fashion has not become reality so far. Therefore, component-based software engineering is still a topic needing significant advances and research.

This chapter is structured as follows: Section 2.1 gives a historical overview from the beginnings of modularization to current plug-in component systems. Section 2.2 defines component terminology relevant for this thesis. Section 2.3 lists the elements of component models. Section 2.4 presents a selection of existing component systems. This selection of systems is analyzed in Section 2.5 with regard to their deficiencies.

2.1 Historical Overview

The history of component-based software goes back to the idea of *modularization*. Modularization aims towards structuring a system in-the-large and the subject of modularization is the design of system architecture. System architecture means the segmentation of a system in components and their interfaces. A component's interface specifies its functional behaviour, its interaction with other components, and its required resources.

Procedural imperative programming improved functional abstraction, but was not adequate for the architecture of large systems, because certain data structures needed to be declared globally. This led to vast interdependencies between procedures and hindered changeability. *Modular programming* improved on these problems by decomposing a system into modules. A module is a compound of algorithms and data structures for a specific task. The usage of a module did not presume any knowledge about the internal structure or about the implementation of the encapsulated algorithms and data structures. A module combined procedures and their shared data into a larger abstract unit. The procedures and data were invisible from the outside, except for those distinguished procedures that represented the module's interface.

Modules are essentially used to implement *abstract data structures*. An abstract data structure is a data structure that is accessed via its interface only. The interface defines what can be

done with the data, and the implementation defines how data and operations are realized. The concept behind abstract data structures is information hiding (Parnas 1972). The goal is to hide implementation details from the user in order to improve changeability. When the data structure is accessible only via the interface, users of the data structure are not affected when the implementation changes. Information hiding also prevents any unintended use of the data structure.

An abstract data structure represents a single instance whereas an *abstract data type* allows the creation of multiple instances, while preserving the benefits of encapsulation. Like abstract data structures, abstract data types use procedures as an interface. However, instead of encapsulating the data in the module, the data are passed as an argument to the procedures. The abstract data type specifies the type's name and interface procedures. How actual variables of this type are structured is hidden from the user. When the internal structure of the type changes, the user of the type is not affected.

In *object-oriented programming* objects encapsulate data like any abstract data type does. Objects are a combination of data and operations that can be performed on the data. The basic premise of object-orientation is to construct programs from sets of interacting and collaborating objects. When in procedural programming a programmer calls a procedure, he specifically determines which algorithm is executed. In object-oriented programming different objects can have different algorithms for the same operation. A method call sends a message to an object. Which algorithm gets called is determined at run time. The user of an object does not consider the differences.

Component-based technology can be seen as an evolution of object-oriented technology (Meyer and Mingins 1999). Nearly all modern component models are based on the object-oriented programming paradigm. However, the concepts of components and objects are independent. The premise of interacting and collaborating objects does not change with components. Like classes, components define object behavior and make their functionality available through interfaces. The most important distinction is that components conform to standards defined by a component model (Weinreich and Sametinger 2001).

Originally appearing in Web browsers, *plug-in components* and systems represent an interesting and promising approach for providing reusable building blocks. An application can be extended by plugging in components at startup time or even at run time. The plug-ins are integrated seamlessly into the system. Eclipse (Eclipse 2003) is the most outstanding representative of these systems and has driven the idea to its extreme ("Everything is a plug-in!" (Beck and Gamma 2003)). Surprisingly, Eclipse has succeeded where previous approaches have failed, namely in building a real component market. A huge community of developers and software vendors has committed itself to Eclipse as the technological basis for developing reusable components and thousands of Eclipse plug-ins can be found on the Web.

2.2 Component Terminology

A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard (Councill and Heineman 2001). A *component model* defines specific interaction and

composition standards. A *component model implementation* is the dedicated set of executable software elements required to support the execution of components that conform to the model. A *software component infrastructure* is a set of interacting software components designed to ensure that a software system or subsystem is constructed using those components and that their interfaces will satisfy clearly defined specifications.

The *interaction* standard specifies actions between two or more software elements. The underlying concept of a component is that it has clearly defined interfaces. An *interface* is an abstraction of the behavior of a component. A component *supports* a *provided interface* if the component contains an implementation of all operations defined by that interface. A component needs a required interface if the component requests an interaction defined in that interface and the component expects some other component to support that interface. Clients interact with a component using the component's clearly defined and documented interfaces.

The *composition* standard specifies how two or more software components can be combined, thereby yielding a new component behavior at a different level of abstraction. The characteristics of the new component behavior are determined by the components being combined and by the way they are combined.

A *component model* operates on two levels. First, a component model defines how to construct an individual component. Second, a component model defines how components will communicate and interact with each other. A component model enables composition by defining an interaction standard that promotes unambiguously specified interfaces. The term component assembly includes the different forms in which components are composed, such as wrapping, static and dynamic linking, and plug-and-play.

The *component model implementation* is the dedicated set of executable software elements necessary to support the execution of components within the component model. The component model implementation is typically a thin layer that executes on top of an operating system. Interfaces are typically defined by using an interface definition language and are registered with an interface repository associated with the component model implementation. The component model implementation makes it possible to execute components that conform to the component model.

2.3 Elements of a Component Model

According to Weinreich and Sametinger (2001) a component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment. The component model also defines standards for an associated component model implementation, the dedicated set of executable software entities required to execute components that conform to the model.

- *Interfaces, Contracts, and Interface Definition Languages.* The main purpose of software components is reuse. Black-box reuse is based on the principle of information hiding (Parnas 1972), and relies on interfaces, which are specifications of component behavior. An interface serves as a contract between a component and its clients. An interface specifies the services a client may request from a component, and the component must provide an implementation of these services. Elements of an interface are

names of operations, their parameters, and valid parameter types. Interface specifications are a central element in a component model.

- *Naming.* A global component marketplace requires uniquely identifiable components and interfaces. Name clashes have to be avoided or at least should be unlikely. Thus a standardizing naming schema is a necessary part of a component model.
- *Meta Data.* Meta data is information about interfaces, components, and their relationships. This information provides the basis for scripting and is used by composition tools. A component model must specify how meta data is described and how it can be obtained. Component model implementations must provide dedicated services allowing the meta data to be retrieved.
- *Interoperability*. Software composition is possible only if components from different vendors can be connected and are able to exchange data and share control through well-defined communication channels. Component interoperability or wiring standards are thus a central element of any component model.
- *Customization*. Customization is the ability of a user to adapt a component prior to its installation or use. Since components are treated as black-boxes, they can only be customized using clearly defined customization interfaces. Customization tools may learn about the customization interfaces of components using meta data services.
- *Composition*. Component composition or assembly is the combination of two or more software components that yields new component behavior. A component composition standard supports the creation of a larger structure by connecting components within an existing structure. Such a structure is a component infrastructure, sometimes called a component framework. The components within a component infrastructure interact with each other, typically through method invocations. The two basic types of component interactions are client/server and publish/subscribe. Components act as clients, calling methods in other components. A component may register itself with another component to receive notifications. The component model must define how to design interfaces to support such composition. Component frameworks enable not only reuse of individual components but of an entire design.
- *Evolution.* Component-based systems require support for system evolution. Components might be replaced with newer versions, with a different implementation or modified interfaces. Existing clients of such components should be affected as little as possible. Rules and standards for component evolution and versioning are thus an important part of a component model.
- *Packaging and Deployment.* A component model must describe how components are packaged, so they can be independently deployed. A component is deployed, that is, installed and configured, in a component infrastructure. The component must be deployed with anything that will not exist in the component infrastructure.

An important part of a component model is the standardization of the run-time environment to support the execution of components. In object-based component systems this includes the specification of interfaces, object creation, life-cycle management, object-persistence support, and licensing.

2.4 Existing Component Systems

Today, there are several component-based approaches which are quite diverse and partly overlapping. Many approaches cannot be regarded as complete component models, because they focus on specific component model elements. This work focuses on component models for rich client applications. Distributed component technologies, with CORBA, COM+, and EJB (Szyperski 2002) as the prime representatives, have different goals. They provide a wiring and interaction standard and a run-time infrastructure for distributed computing and client/server systems.

JavaBeans (Sun 1996) have extended object-oriented programming technology by concepts for clear interface specification, component customization at build time and component deployment. The Java Service Loader (Sun 2006) extends the interaction standard, adds meta data, and allows composition at startup time. The Java Service Loader implements a locator for Java services. A Java service is a set of interfaces and abstract classes. A service provider is a specific implementation of a service. The classes in the provider typically implement the interfaces and subclass the classes defined in the service itself. Service providers can be installed in an implementation of the Java platform in the form of extensions, i.e. jar files placed into any of the usual extension directories. Providers can also be made available by adding them to the application's class path.

The NetBeans Lookup API (Boudreau et al. 2007) is another example for a service locator. The client side lookup works similar to the Java Service Loader, the provider side actually uses the same Java services as the Service Loader. The NetBeans Lookup API offers a general registry permitting clients to find instances of services. The distinguished feature of the Lookup API is that it is designed with dynamic change in mind.

The PicoContainer (Pico 2009) is a well-known implementation of the dependency injection pattern (Fowler 2004). The basic idea of an inversion of control container is that the container is a separate object, that actively resolves dependencies between components. During composition, when a client needs a component, the container searches available provider components, and injects a reference to the provider into the dependent component.

The OSGi framework (OSGi 2006) is a module system for Java that defines a standard for deploying and managing coarse-grained components. The component model defines life-cycle management of components (called bundles in OSGi), a service registry, and an execution environment. In OSGi allows components to be started, stopped or replaced without requiring a reboot. Technically, the OSGi service framework is essentially based on a custom Java class loader and a service registry that is globally accessible within a single Java virtual machine (Hall and Cervantes 2004).

Eclipse (Eclipse 2003) can be regarded as the most outstanding representative of plug-in systems today. Eclipse plug-ins (so-called extensions) are fine-grained components with a well-defined and published interface that can plug into so-called extension points of other components. Extensions and extension points are specified in XML files, which Eclipse uses for discovery and loading of plug-ins. The Eclipse IDE comprises wizards which make it easier for developers to create the XML files.

MagicBeans (Chatley et al. 2004) is a Java framework which enables automatic component assembly at run time. It relies on reflection to inspect Java class files and to match and bind interface definitions and corresponding class implementations. The MagicBeans component model is simplistic in the sense, that it does not support meta data that can be used for composition, or that it cannot distinguish between extension points that share the same interface.

The MADAM system presented in (Hallenstein et al. 2006; Floch et al. 2006) is another interesting approach. It uses dynamic discovery of components to support system adaptation of mobile devices to changing environmental conditions such as available bandwidth or network connectivity. It uses an architecture variability model to guide system adaptation and reconfiguration.

2.5 Deficiencies of Existing Component Systems

In the problem statement (see page 3) we argued that rich client application should be customizable, extensible, and reconfigurable at run time. In existing component systems, we see two major shortcomings with regard to customizability and dynamic change:

- *Lack of granularity.* Fine-grained composition options are required to recompose an application for each working situation. Thus the feature set is always aligned to the task at hand. Existing component systems are designed for coarse-grained composition. Existing component systems only allow to replace plug-ins as a whole. They do not allow to keep some parts of a plug-in, while replacing other parts with custom extensions. The composition operations in existing component systems have system-wide affect, because interface registries deliver change notifications system-wide. Such systems cannot change components only in specific parts of the system.
- *Lack of dynamic reconfiguration support.* Support for dynamic change is required to reconfigure an application on the fly by swapping sets of components. Reconfiguration means to dynamically add, remove, or replace components. In existing component systems, support for dynamic change is weak mainly for three reasons: Firstly, the client alone is responsible to compose the providing components programmatically. The component infrastructure has no control over whether, how, or when a client looks for components. Secondly, clients typically compose only at startup. Support for dynamic change in existing component infrastructures is optional, if supported at all. Eventually, some components are aware of dynamic change, but most components are not, thus hindering pervasive support for dynamic change. Thirdly, existing component models have different API's for startup composition and for dynamic change. That obligates programmers to provide two different implementations.

Weinreich and Sametinger (2001) use an analogy of operating systems to clarify the shortcomings. In the analogy, operating systems are component model implementations for applications, which may be viewed as coarse-grained components. Once a component model implementation is developed, multiple vendors can develop applications that use the services provided by the component model implementation. Components at the application level can be used, but they insufficiently enable widespread reuse. The lack of reuse occurs because applications are too coarse-grained and they lack composition support. This is also true, if we

understand reuse in the context of dynamic reconfiguration within the same application. When we want to reconfigure for a specific configuration, the coarse-grained model is a problem.

In order to improve reuse, Weinreich and Sametinger define several goals for component-based software engineering. Two of those goals are relevant for this thesis:

- Applications are to coarse-grained to improve software reuse. Application developers are often required to design and implement common functionality that any application may have. Component-based software engineering seeks to factor out these commonalities into either services provided by the component model implementation or components that could be purchased and integrated into a component infrastructure. A central concept of component-based software engineering is to develop technologies for smaller, fine-grained components and enable a similar degree of reuse on the level of application parts as was possible at the application level.
- While applications have long been units of independent deployment, there has typically been no support for composition. In fact, operating systems ensure that applications execute in complete isolation from each other. While applications deploy in the operating system, they are rarely units of composition. The goal of component-based software engineering is to develop systems by composing reusable components at a finer level of granularity than applications.

In the remainder of this section we show what existing component systems offer with regard to customizability and dynamic change, and what limitations they have.

2.5.1 Lack of Granularity

Granularity in a component model matters two-fold. Firstly, when a component model defines how components are constructed, it defines the construction granularity. And when a component model defines how components are assembled, it defines the composition granularity. Secondly, a component model defines what parts of an application are affected by composition, thereby defining the scope of a change operation.

As a component model is about components, it allows composition with component granularity. For more fine-grained customization options, a more fine-grained granularity would be useful. When a component provides multiple interfaces, the component model could allow composing individual interfaces. Table 1 (on page 18) gives an overview how existing component systems construct components, and what granularity they support in composition.

A component model defines what parts of an application are affected by composition, thereby defining the scope of a change operation. In existing component systems change operations that add or remove a component have system-wide effect. A more fine-granular model could allow adding a component only to specific parts of an application, or removing a component only from specific parts of an application.

JavaBeans, NetBeans, and MagicBeans (Sun 1996, Boudreau et al. 2007, Chatley et al. 2004) offer the same level of granularity. In these models, a component is a JavaBean, a Java object that is composable by following specific naming conventions. Multiple JavaBeans can be

combined and deployed as a Jar file. When deployed as a Java service, the Service Loader (Sun 2006) allows composing Jar files. Composition scope is always system-wide.

The Pico-Container (Pico 2009) is a representative for inversion of control containers. In Pico, the component is a Java class implementing a specific interface. Before assembly, the components are installed into a Pico container. The container allows composition on an interface-level. Pico also supports scoped containers with parent/child relationships as unit of composition.

Component System	Component Construct	Construction Granularity	Composition Granularity	Composition Scope
JavaBeans/NetBeans/ MagicBeans	Bean	Bean Jar	Jar	System-wide
PicoContainer	Component	Component	Interface Container	Container System-wide
OSGi	Bundle	Service Bundle	Service Bundle	System-wide
Eclipse	Extension	Extension Plug-in	Extension	System-wide

Table 1. Components and granularity of existing component systems

The OSGi service platform (OSGi 2006) uses so-called bundles as components. A bundle is a Jar file that contains multiple Java classes and meta information. One bundle can register multiple services in the OSGi service registry. Services and bundles are units of composition with system-wide scope.

The Eclipse platform (Eclipse 2003) uses extensions as components. An extension is a class that implements an interface. A plug-in combines multiple extensions in a Jar file together with XML meta data. The sole unit of composition is the extension, since the extension registry has no operations on plug-in level. Independent from the core Eclipse component model, the Eclipse Integrated Development Environment (IDE) adds two coarser elements above the extension. A *feature* combines multiple plug-ins as a unit that can be deployed and installed together. A *product* packages a combination of features into their own instance of the Eclipse IDE. A product can be perceived as an individual application. Composition scope is always system-wide.

2.5.2 Lack of Dynamic Reconfiguration Support

Dynamic reconfiguration requires support in the component model. Composition operations for dynamic addition or removal must be provided in the component model and in the component model implementation. Support for dynamic change is weak in existing component systems for three reasons:

- The client (not the component infrastructure) composes providers programmatically. The component infrastructure has no control over whether, how, or when a client composes providers.

- The client typically composes only at startup, the support for dynamic change is optional. Some components are aware of dynamic change, but most are not, thus hindering pervasive reconfiguration support.
- The component systems offers different programming models for composition at startup and for dynamic change. The component programmer is obligated to provide two different implementations.

The remainder of this section discusses the three reasons in their own subsections. We use a sample movie application to demonstrate in detail the deficiencies of current component systems from the view of a component programmer.

Movie Application Example

This section uses a sample program to discuss the weaknesses in current component systems. The example is intentionally kept simple, but it is sufficient to visualize what the problems are. Fig. 1 shows the static structure of the movie application with three components. The movie application component uses a move lister component that provides a list of movies directed by a particular director. The point we focus on in this example is where the movie lister component uses one or more finder components. The movie lister asks each finder component to return every movie it knows about.

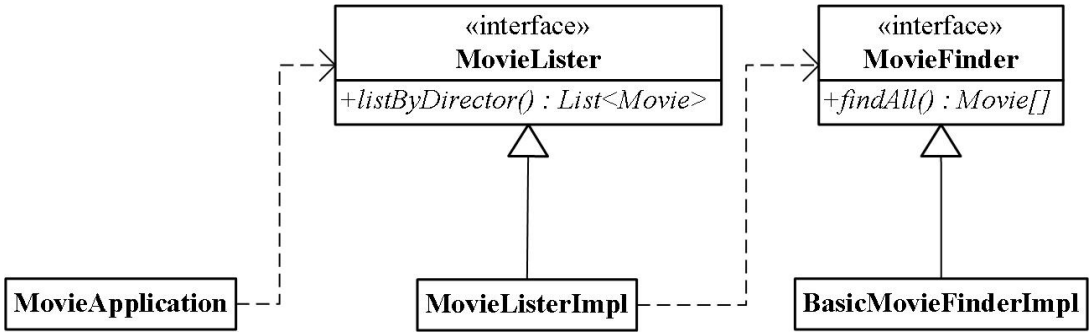


Figure 1. Class diagram of movie application example

Our starting point is an object-oriented Java implementation of the movie application. The application comprises the three components *MovieApplication.jar*, *MovieLister.jar*, and *BasicMovieFinder.jar*. The components are decoupled by the *MovieLister* and *MovieFinder* interfaces. All finder components know the `findAll` method.

```
public interface MovieFinder {
  Movie[] findAll();
}
```

The implementation of the `listByDirector` method in class `MovieListerImpl` is straightforward. The method uses an array of interface `MovieFinder` to keep references to provider components. We will see later how each component system fills that array. For the moment, we take the provider array as given. The movie lister gets an array with movies from each provider and filters for the director's name.

```
public List listByDirector(String director) {
  private MovieFinder[] finderArray = ...;
  List result = new LinkedList();
```

```
    for(MovieFinder finder: finderArray)
       for(Movie movie: finder.findAll())
         if(movie.getDirector().indexOf(director) > -1)
           result.add(movie);
    return result;
  }
```

With the `MovieFinder` interface, we have achieved that lister and finder are decoupled. However, at some point we have to come up with a concrete class for the finder. The following sections show how to do this in the respective component system. They also discuss why this is a problem for dynamic change.

Provider integration requires programming effort

The first reason why existing composition systems are weak with regard to dynamic reconfiguration is that composition is done programmatically inside the client component. Thus it is the programmer's responsibility to control how the composition is performed. Most systems have some kind of interface registry where components register their services. The client queries the registry to look for providers. Typically the client does this only at startup. Support for dynamic change is optional, if supported at all.

To complete our plain Java implementation from above, we statically bind the provider implementation at compile time in the array constructor. We statically bind two provider classes, `BasicMovieFinderImpl` and `FileMovieFinderImpl`.

```
private MovieFinder[] finderArray = new MovieFinder[] {
  new BasicMovieFinderImpl(),
  new FileMovieFinderImpl()
};
for(MovieFinder finder: finderArray) { ... }
```

The problem with this implementation should be obvious. The movie lister component is statically bound to the two providers specified in the source code. There is no way for a user to change the configuration.

Java Service Loader

The Java Service Loader (Sun 2006) solves the compile time dependency problem. The lister uses the movie finder *service*, which is specified by the `MovieFinder` interface. The `load` method of the service loader class dynamically discovers available service providers. A *service provider* is a specific implementation of a service. The service loader loads providers on demand and caches instances for later access.

```
ServiceLoader<MovieFinder> loader = ServiceLoader.load(MovieFinder.class);
for(MovieFinder finder: loader) { ... }
```

The discovery mechanism uses a simple configuration file. The name of the file designates the fully qualified service name, and the file contains a list of fully qualified names of concrete provider classes. If for example `movies.impl.BasicMovieFinderImpl` is an implementation of the `MovieFinder` service, then its Jar file contains a file named:

```
META-INF/services/movies.MovieFinder
```

This file contains the single line:

```
movies.impl.BasicMovieFinderImpl
```

The service loader decouples service user and service provider. It discovers service providers at run time. However, the lister client has to compose its providers programmatically. The typical implementation does this only once at startup. It is not possible to change providers thereafter.

NetBeans Lookup

NetBeans Lookup (Boudreau et al. 2007) solves the dependency problem similar to the Java Service Loader. On the client side, the classes in package Lookup work similar to the service loader. The lister client queries the default lookup implementation for providers. The registration of service providers is exactly the same as with the Java Service Loader.

```
Lookup.Template template = new Lookup.Template(MovieFinder.class);
final Lookup.Result result = Lookup.getDefault().lookup(template);
...
Collection<? extends MovieFinder> finders = result.allInstances();
for(MovieFinder finder: finders) { ... }
```

With NetBeans Lookup, the client programmatically composes in a similar way as the Java Service Loader does. The shown implementation is default in NetBeans. It does not support dynamic change. However, NetBeans Lookup optionally offers support for dynamic change (see page 25).

PicoContainer Dependency Injection

An *Inversion of Control* container solves the compile time dependency problem by moving the composition code outside of the client. The container uses the *Dependency Injection* pattern to inject the provider object into the client object. In a Pico application, the lister class has an array of finders, and a constructor with the same array as a parameter. In this example, we use constructor injection to set the dependency via the object's constructor. Pico supports a slew of other injection mechanisms, for example field or method injection (Pico 2009). All the lister has to do, is to provide a constructor with arguments for its dependencies.

```
public class MovieListerImpl implements MovieLister {
  private final MovieFinder[] finders;
  public MovieListerImpl(MovieFinder[] finders) {
    this.finders = finders;
  }
  public List listByDirector(String director) {
    ..
    for(MovieFinder finder: finders) { ... }
    ..
  }
}
```

The composition code is outside of the client component, for example in the movie application component. In the `main` method, we create a new pico container and add all components to the container. When we call `getComponent` on the container to create the lister, the Pico container injects instances of all classes which are assignable to the `MovieFinder` interface in the movie lister constructor. Thus, the movie lister is configured with the `BasicMovieFinderImpl` class and the `FileMovieFinderImpl` class.

```
public static main(String[] args) {
  pico = new DefaultPicoContainer(...);
  pico.addComponent(MovieApplication.class);
  pico.addComponent(MovieListerImpl.class);
  pico.addComponent(BasicMovieFinderImpl.class);
  pico.addComponent(FileMovieFinderImpl.class);
  pico.start();
  MovieLister lister
    = (MovieLister) pico.getComponent(MovieLister.class);
  lister.listByDirector("Ang Lee");
}
```

The inversion of control container decouples service client and provider. It moves the composition code away from the client component to some main part of the application. The inversion of control container constitutes a separate composer instance responsible for composition and thereby offers a solution for the problem of programmatic integration. However, the Pico container also composes only at startup. It does not allow changing the composition at run time.

OSGi Service Registry

The OSGi service registry (OSGi 2006) solves the dependency problem similar to the Java Service Loader. The lister uses a tracker to get all registered service providers. The tracker delivers references to all registered services.

```
ServiceTracker tracker = new ServiceTracker(...,
    MovieFinder.class.getName(), null);
tracker.open();
for(ServiceReference ref: tracker.getServiceReferences()) {
  MovieFinder finder = (MovieFinder) tracker.getService(ref);
  ...
}
```

Unlike the Java service loader, OSGi bundles programmatically register their services in the service registry. The bundle registers the service when the run-time environment loads the bundle. The service registry identifies a service by its name. Optionally, the provider can register a dictionary with properties. During composition, the client can use those properties to filter service providers.

```
public void start(BundleContext context) {
  MovieFinder finder = new BasicMovieFinder();
  context.registerService(MovieFinder.class.getName(), finder, null);
}
```

The service registry decouples service client and service provider. However, the client itself composes programmatically. The provider also has to provide code to register a service in the registry. When hosts use this simple programing model, it is not possible to dynamically change services from outside. The OSGi service registry optionally offers a different model with support for dynamic change (see page 26).

Eclipse Extension Registry

Eclipse (Eclipse 2003) builds on an OSGi implementation called Equinox. However, Eclipse uses its own Extension Registry instead of the OSGi service registry. As far as provider inte-

gration is concerned, both work similar, but we will see later, that they are different when it comes to dynamic reconfiguration.

The lister uses the extension registry to get a reference to the extension point. The extension point retrieves a collection of available extensions. The mechanism for selecting the desired provider from the list of all retrieved extensions is cumbersome. The client must search the configuration elements of every extension for the desired class name before he can create an instance of the provider class.

```
IExtensionPoint point = Platform.getExtensionRegistry()
    .getExtensionPoint("at.jku.ase.MovieFinder");
for(IExtension ext: point.getExtensions()) {
  IConfigurationElement element;
  for(IConfigurationElement e: ext.getConfigurationsElements()) {
    if(e.getName().equals("MovieFinder")) {
      elem = e;
      break;
    }
  }
  MovieFinder finder = elem.createExecuteableExtension("class");
  ..
}
```

In contrast to the programmatic approach in the OSGi service registry, the registration of extensions in the Eclipse extension registry is declarative. Each extension provides a *plugin.xml* configuration file. This file specifies the target extension point, and the class implementing the service.

```
<plugin>
  <extension point="at.jku.ase.MovieFinder">
    <MovieFinder class="movie.impl.BasicMovieFinderImpl"/>
  </extension>
</plugin>
```

The extension registry decouples client and provider extension through an extension point. The provider uses declarative specification, but the client has to compose the extensions programmatically, typically at startup. Using the shown implementation, it is not possible to dynamically change providers at run time. Optionally, the extension registry offers a different mechanism for dynamic reconfiguration (see page 27).

Summary

The mechanism how a client integrates providers is similar in most of the described component systems. A client component provides code that queries some kind of registry at startup. The fact that the client component controls when or how it integrates provider components, and the circumstance that this is done typically only at startup, is one reason why dynamic reconfiguration is a problem.

Dependency injection is an exception here. The dependency injection container uses constructor injection to wire up client and provider. The client component does not provide code for contributor composition.

Provider integration requires programmatic effort	
hard-wired Java application	Client queries collection of contributor classes
Java Service Loader	Client queries service loader
NetBeans Lookup	Client queries lookup
Pico Dependency Injection	Container automatically injects provider
OSGi Service Registry	Client queries service tracker
Eclipse Extension Registry	Client queries extension registry

Table 2. Programmatic provider integration in existing component systems

Dynamic change support is optional

The second reason why existing composition systems are weak with regard to dynamic reconfiguration is that composition systems primarily focus on composition at startup. When we look at dynamic change, composition systems can be subdivided into two groups. One group lacks support for dynamic change, and the other group has some support for it, but whether a component supports dynamic change is optional. In practice, when dynamic change awareness is optional, some components do support it, and others don't. This is a problem, because it hinders pervasive support for dynamic change.

When in an application some components are dynamic-aware and others are not, the resulting problems differ depending on the kind of change. When we dynamically add a component, those components that are aware, will integrate the new functionality, while those components that are not aware, will ignore it. The composition result might at least be partly usable. When we dynamically remove a component, the consequences are more severe. If there are components in the system that do not stop using the removed component, the component cannot effectively be removed.

Of course, our statically bound Java version of the movie application does not support dynamic change. The dependency to the finder providers was introduced at compile time and cannot change.

```
private MovieFinder[] finderArray = new MovieFinder[] {
  new BasicMovieFinderImpl(),
  new FileMovieFinderImpl()
};
for(MovieFinder finder: finderArray) { ... }
```

Java Service Loader

The Java service loader (Sun 2006) class has operations to manually load or reload the collection of service providers. However, the client is not notified when the set of available service providers changes. The only way to update the dependencies is to periodically reload the set of service providers and determine added or removed providers by calculating the difference between two consecutive calls.

```
ServiceLoader<MovieFinder> loader = ServiceLoader.load(MovieFinder.class);
for(MovieFinder finder: loader) { ... }
```

The service loader does not support dynamic change, because there is no change notification mechanism.

NetBeans Lookup

NetBeans Lookup (Boudreau et al. 2007) offers a mechanism for dynamic change notification. The example uses a lookup template to request providers for the movie finder interface. The result of the lookup request allows registering a change listener. The lookup result can call back the change listener's `resultChanged` method when providers are added or removed.

```
public List listByDirector(String director) {
  Lookup.Template template = new Lookup.Template(MovieFinder.class);
  final Lookup.Result result = Lookup.getDefault().lookup(template);
  result.addLookupListener(new MyListener());
  ...
  Collection<? extends MovieFinder> finders = result.allInstances();
  for(MovieFinder finder: finders) { ... }
}

class MyListener implements LookupListener {
  public void resultChanged(LookupEvent e) { ... }
}
```

However, the information the client component gets when the lookup result changes is sparse. The lookup just advises the client that the result has changed, with not additional information about the kind of change. The client component cannot determine whether something has been added, or removed, nor does it know which providers are affected by the change. As a response to that change, the client can either recompose all contributors, or if it wants to selectively add or remove components, it can compare the states before and after the change manually.

The NetBeans Lookup discovers the Java extensions in the same way as the Java Service Loader API. In addition to that, the NetBeans platform allows programmers to specify a change listener. However, none of the lookup providers that come with NetBeans does support this listener. To implement dynamic change in NetBeans, a programmer still has to provide his own implementation of a lookup provider.

PicoContainer Dependency Injection

Inversion of control containers like Pico (Pico 2009) are designed to inject dependencies once at startup. Once the dependency is injected, it can neither be rejected from the container, nor can further providers be added.

In the sample code below, we add the lister and the basic movie finder to the composition container. Then we call `getComponent` to create the lister component with the basic movie finder as the only provider. Each `addComponent` or `removeComponent` call after creation of the lister will effect newly created listers, however it cannot affect already composed components.

```
MutablePicoContainer pico = new DefaultPicoContainer(...);
pico.addComponent(MovieApplication.class);
pico.addComponent(MovieListerImpl.class);
pico.addComponent(BasicMovieFinderImpl.class);
MovieLister lister = (MovieLister) pico.getComponent(MovieLister.class);

pico.removeComponent(BasicMovieFinderImpl.class);
pico.addComponent(FileMovieFinderImpl.class);
// does not affect 'lister'
```

The dependency injection model does not support dynamic change. The constructor injection mechanism is by definition made for startup composition only.

OSGi Service Registry

The OSGi service registry (OSGi 2006) is designed to start and stop services dynamically. If a client component wants to support dynamic change, it implements a service tracker. The client component subclasses the service tracker and overrides the methods `addingService` and `removedService`.

```
class MovieFinderTracker extends ServiceTracker {
  // Collection finders declared in outer class
  public MovieFinderTracker(BundleContext context) {
    super(context, MovieFinder.class.getName(), null);
  }
  public Object addingService(ServiceReference reference) {
    MovieFinder finder = (MovieFinder) context.getService(reference);
    finders.add(finder);
    return finder;
  }
  public void removedService(ServiceReference reference, Object service) {
    MovieFinder finder = (MovieFinder) service;
    context.ungetService(reference);
    finders.remove(finder);
  }
}
```

The specialized movie finder tracker is an inner class in the movie lister implementation. The tracker automatically keeps the collection with the movie finder providers up to date.

```
public class MovieListerImpl implements MovieLister {
  private final MovieFinderTracker tracker;
  private Collection finders =
    Collections.synchronizedCollection(new ArrayList());
  public MovieListerImpl(BundleContext context) {
    tracker = new MovieFinderTracker(context);
  }
  public ServiceTracker getTracker() { return tracker; }
  public List listByDirector(String director) {
    MovieFinder[] finderArray = (MovieFinder[])
      finders.toArray(new MovieFinder[finders.size()]);
    for(MovieFinder finder: finderArray) { ... }
    ...
  }
}
```

The code in the `BundleActivator` below shows how to add or remove services to the service registry. The `registerService` method in class `BundleContext` registers a service object

under the specified interface name into the service registry. The `unregister` method in interface `ServiceRegistration` unregisters the service object from the service registry. After a service has been unregistered, associated `ServiceReference` objects can no longer be used to interact with the service.

```
public class BasicMovieFinderActivator implements BundleActivator {
  private ServiceRegistration registration;
  public void start(BundleContext context) {
    MovieFinder finder = new BasicMovieFinderImpl();
    registration = context.registerService(
        MovieFinder.class.getName(), finder, null);
  }
  public void stop(BundleContext context) {
    registration.unregister();
  }
}
```

The service registry supports dynamic change, however, whether a client uses a sub-classed service tracker is optional.

Eclipse Extension Registry

The Eclipse extension registry (Eclipse 2003) allows adding extensions dynamically. If a host component wants to support dynamic change, it implements an extension change handler. The handler gets change notifications for a given extension point. The `addExtension` method is called whenever an extension is added. The `removeExtension` method is called whenever an extension is removed.

```
IExtensionRegistry reg = Platform.getExtensionRegistry();
IExtensionPoint xp = reg.getExtensionPoint("at.jku.ase.MovieFinder");
IExtensionTracker tracker = new ExtensionTracker(reg);
IFilter filter = ExtensionTracker.createExtensionPointFilter(xp);
IExtensionChangeHandler h = new MyHandler();
tracker.registerHandler(h, filter);

class MyHandler implements IExtensionChangeHandler {
  public void addExtension(IExtensionTracker t, IExtension ext) {
    // read configuration and create extension executable extension
  }
  public void removeExtension(IExtension ext, Object[] objects) {
    // release references to extension
  }
}
```

The extension registry has methods for adding and removing entries from the registry. The `addContribution` method adds extension points, extensions or a combination of those described in a *plugin.xml* file. The `removeExtension` method removes an extension from the registry. Those calls trigger the notification of change handlers like the one shown above.

```
IExtensionRegistry reg = Platform.getExtensionRegistry();

// add an extension
Bundle bundle = Activator.getDefault().getBundle();
IContributor contributor = ContributorFactoryOSGi.createContributor(bundle);
InputStream stream = new FileInputStream("plugin.xml");
reg.addContribution(stream, contributor, false, null, null, null);
stream.close();
```

```
// remove an extension
IExtension ext = ...;
reg.removeExtension(ext, null);
```

The Eclipse methods for dynamic change are part of an interim API that is still under development. At this early stage, it is made available to solicit feedback. The primary composition mechanism in Eclipse is still startup composition. Support for dynamic change is optional, and to date almost no plug-in does it. This fact can be easily observed in the Eclipse IDE, because when a feature is removed, the IDE requires a restart.

Summary

The support for dynamic change varies among the component systems described in this chapter. The Java Service Loader and Pico Dependency Injection are limited to startup composition. NetBeans Lookup has dynamic change in the API, however, none of the lookup providers that come with NetBeans allows triggering changes from outside. The Eclipse extension registry does support dynamic change, but since dynamic change support is optional, most Eclipse plug-ins do not support it. Finally the OSGi service registry supports dynamic change and services can be added and removed freely. The unpleasant thing with OSGi is that dynamic change support requires a lot of code to write.

Dynamic change support is optional	
hard-wired Java application	no dynamic change
Java Service Loader	no dynamic change
NetBeans Lookup	API for dynamic change support, but not implemented in NetBeans lookup providers
Pico Dependency Injection	no dynamic change
OSGi Service Registry	dynamic change support, but optional
Eclipse Extension Registry	experimental dynamic change support, plug-ins do not support it

Table 3. Optional dynamic change support in existing component systems

Non-uniform programming model for startup and dynamic change

The previous two sections explained two problems for dynamic change. Firstly, the component infrastructure cannot trigger change operations dynamically, because components will not react. Client components programmatically compose providers at startup and do not change their composition thereafter. Secondly, even if a component system supports dynamic change, it makes support for dynamic change optional. Thus, even if the component system supports dynamic change, most components do not.

This section discusses a third problem which is sort of a combination of the other two. Existing component systems use different programming models for initial composition at startup, and for dynamically adding or removing components. This is a problem because component programmers are obliged to provide two different implementations.

OOP, Java Service Loader, and Pico Dependency Injection are excluded from this section, because they do not support dynamic change at all.

NetBeans Lookup

NetBeans Lookup (Boudreau et al. 2007) distinguishes startup composition and dynamic change. At startup, the client component queries the lookup repository for all matching instances of providers. In order to react to dynamic change, the client component additionally registers a change listener.

```
Lookup.Template template = new Lookup.Template(MovieFinder.class);
final Lookup.Result result = Lookup.getDefault().lookup(template);
// do startup composition
Collection<? extends MovieFinder> finders = result.allInstances();
for(MovieFinder finder: finders) { ... }
// register change listener
result.addLookupListener(new MyListener());

class MyListener implements LookupListener {
  public void resultChanged(LookupEvent e) {
    // react to change
  }
}
```

The code in the change listener is completely different from the code for integrating components at startup. Since there is no information about the kind of change, the client component has to manually find out changes by comparing old and new state.

OSGi Service Registry

The OSGi service registry (OSGi 2006) actually is the positive exception, because the subclassed service tracker handles startup composition and dynamic change in a uniform model (see page 26). A small caveat is that support for dynamic change is still not mandatory. A client component can choose between the special service tracker with dynamic support, and the basic service tracker without dynamic support (see page 22).

Eclipse Extension Registry

The Eclipse extension registry (Eclipse 2003) distinguishes startup composition and dynamic change. At startup, the client component queries the extension point for initial extensions. In order to react to dynamic change, the client component registers a registry change handler.

```
IExtensionPoint point
  = Platform.getExtensionRegistry()
      .getExtensionPoint("at.jku.ase.MovieFinder");
for(IExtension ext: point.getExtensions())
  addExtension(ext);

IExtensionTracker tracker = new ExtensionTracker(reg);
IFilter filter = ExtensionTracker.createExtensionPointFilter(xp);
IExtensionChangeHandler h = new MyHandler();
tracker.registerHandler(h, filter);

class MyHandler implments IExtensionChangeHandler {
  public void addExtension(IExtensionTracker t, IExtension ext) {
    addExtension(ext);
  }
```

```
    public void removeExtension(IExtension ext, Object[] objects) {
      removeExtension(ext);
    }
  }
  private void addExtension(IExtension ext) { ... }
  private void removeExtension(IExtension ext) { ... }
```

The change listener represents a different model for composition than the initial query at start-up. The programmer of the client component has to provide two implementations.

Summary

Among those component systems that support dynamic change, all but one use separate mechanisms for startup and dynamic changes. Typically, there is some kind of central provider registry. When a component is activated, it gets a collection of available providers from the registry. This comprises the initial composition. Then the host component has to provide an additional implementation. Typically it registers a listener in the registry to handle dynamic change. In all systems the implementation for dynamic change is optional and can be omitted by a client component. The consequence is that some components support dynamic change, while others do not.

Non-uniform programming model for startup and dynamic addition

hard-wired Java application	no dynamic change
Service Loader	no dynamic change
Lookup	separate code for startup and dynamic change
Dependency Injection	no dynamic change
Service Registry	uniform model
Extension Registry	separate code for startup and dynamic change

Table 4. Non-uniform programming models in existing component systems

Chapter 3: Plux.NET Composition Model

This chapter describes a composition model (CM) which addresses the deficiencies described in Section 2.5. The CM is the foundation for the composition infrastructure described in Chapter 4. The CM defines meta elements for components and their relationships, and it specifies services for discovery, qualification, and composition.

This chapter is structured as follows: Section 3.1 describes key characteristics of the Plux.NET approach. Section 3.2 describes how Plux.NET is related with the .NET Framework. Section 3.3 describes the Plux.NET CM based on the metaphor of slots and plugs. Section 3.4 describes the meta elements in the CM. Section 3.5 to 3.7 describe the discovery service, the qualification service, and the composition service. Section 3.8 describes the lifecycle of meta elements. Section 3.9 describes how the discovery, qualification and composition service compose an application.

3.1 Characteristics of the Plux Approach

Plux.NET is a plug-in component framework focussing on the concept of dynamic composition. Like existing plug-in systems, Plux.NET is based on the concept of a small core that is extended with plug-in components. Plug-ins can be plugged into the core or into other plug-ins where they are integrated seamlessly by their host. This plug-in component foundation allows developers to build extensible and customizable applications with Plux.NET.

Unlike in existing systems, the plug-in discovery mechanism is not an integral part of the Plux.NET composition infrastructure. Instead, the composition infrastructure can be extended with custom discovery plug-ins. The discovery plug-in is responsible for detecting additions or removals of plug-ins. In Plux.NET, discovery is designed for dynamic change, allowing addition and removal of plug-ins at run time. Discovery plug-ins themselves can also be dynamically replaced, thus allowing applications to use different discovery mechanisms for launching and while they run. Since the discovery plug-in is also responsible for meta data provision, the meta data mechanism can be replaced.

Unlike existing plug-in systems, the Plux.NET composition infrastructure does not operate as a passive registry where components themselves drive composition. Instead, the composition infrastructure comprises a *composer* which actively controls composition from the core. The

components have a passive role in the composition process, i.e., they are controlled by the composer.

The active composer is a precondition for *plug-and-play composition*. The composition infrastructure specifies that components *declare* their requirements and provisions with *meta data*. The composer uses the composition infrastructure to obtain those meta data, and tries to match requirements and provisions. It creates component instances and notifies host components that a contributor component became available. The composition infrastructure *stores* which host component *uses* which contributor component. The composer uses these stored connections when it needs to notify host components that a contributor component becomes unavailable. This is done before it releases a component instance. The host components strictly adhere to an *event-based* programming model. They react to the event notifications of the composer and dynamically add or remove contributor components. This plug-and-play approach allows composing applications without programming.

The composition infrastructure supports an automatic and a manual mode for plug-and-play composition. In *automatic* mode, the composer integrates any plug-in which becomes available. This makes composing applications as simple as dropping plug-ins in the application directory. In *manual* mode, composition tools use the public interface of the composer to connect plug-in components.

3.2 Prerequisites for Plux.NET

The .NET moniker in the name Plux.NET suggests a connection between Plux.NET and the .NET Framework. This connection should be understood as "the version of Plux for .NET", because the CM is not tied to .NET, and could, for example, be adapted for the Java Runtime Environment.

The reason why the Plux.NET composition infrastructure requires a base technology such as the .NET Framework is that Plux.NET does not specify all elements of a complete component model (see page 13). The Plux.NET CM specifies standards for interfaces and contracts, naming, meta data, customization, and composition. It does not specify standards for interoperability, evolution, packaging, and deployment. Instead, Plux.NET declares those elements as prerequisites. The Plux.NET composition infrastructure as described in Chapter 4 meets the prerequisites with capabilities of .NET. The .NET Common Language Infrastructure (CLI) provides standards for component interoperability, evolution, packaging, and deployment (ECMA 2006).

The choice of .NET as a platform for the Plux.NET composition infrastructure is primarily motivated by our industry partner's willingness to participate in case studies and pilot projects, provided that the base technology is compatible with his ERP applications (Reiter 2007, Wolfinger 2008a, Rabiser 2009).

3.3 Composition with Slots and Plugs

The Plux.NET CM uses the metaphor of extensions with slots and plugs. It composes an application of *extensions* with well-defined interfaces. An extension is a functional component which provides services to other extensions or uses services provided by other extensions. As Fig. 2 shows, an extension opens a slot when it wants to use the service of other extensions, and it provides a plug when it provides a service to other extensions. Non-trivial extensions can have multiple slots and plugs. The CM defines the mediating process which matches required and provided services, or in other words, which composes an application by plugging plugs into slots.

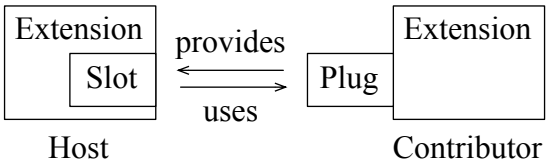


Figure 2. Slot and plug in host and contributor extension

A *slot* specifies how other extensions are intended to extend the functionality of this extension, whereas a *plug* specifies how this extension makes contributions to others. Therefore, slot and plug specifications have to match. In essence, a slot declares the kind of information an extension expects and the plug fills these information slots. Accordingly, an extension which opens a slot is called *host extension,* whereas an extension filling a slot is called *contributor extension*. Contributor extensions again can open their own slots where other extensions can contribute allowing the whole application to grow.

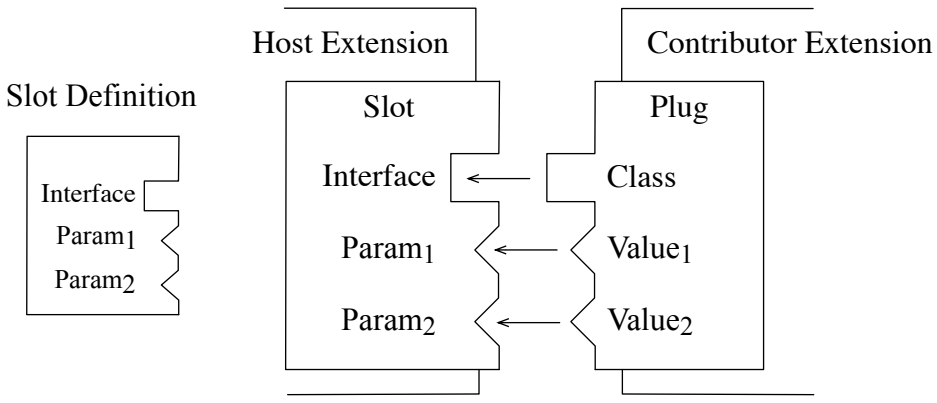


Figure 3. Slot definition in host and contributor extension

Contributions occur on the level of run-time behavior, i.e. host and contributor will communicate based on a defined protocol to accomplish a particular task. The collaboration between host and contributor is defined by a *slot definition* in the form of a required and a provided *interface*. The host defines the required interface and the contributor has to provide an implementation for it. Fig. 3 shows a slot definition which specifies the structure of a slot in the host extension and the corresponding plug in the contributor extension. The interface in the host and the implementing class in the contributor constitute the agreed collaboration protocol. Additional *parameters* and their *values* define other properties that the host requires to

make use of the extension. The host defines required parameters and the contributor has to provide values for them.

Functionally related extensions are packed in a *plug-in* and thus can be installed jointly. Similarly, functionally related slot definitions are packed in a *contract*. Slot definitions are kept separate from the extensions, because they need to be published to contributors without also publishing the extensions that open these slots.

3.4 Meta Elements

The CM uses meta data to describe extensions and their relationships. *Type meta elements* describe the static elements of contracts and plug-ins. Fig. 4 shows that contracts contain slot definitions, and slot definitions contain parameter definitions. Plug-ins contain extension types, extension types contain slot types and plug types, and plug types contain parameters. *Instance meta elements* describe instances of composed extensions which are connected via their slots and plugs.

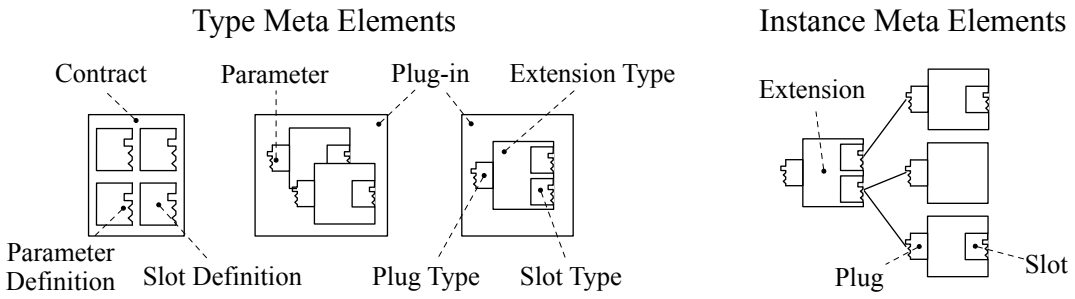


Figure 4. Plux.NET composition model meta elements

For better readability of the text in the remainder of this chapter, we refer to type meta elements shortly as *types*, and to instance meta elements shortly as *instances*, wherever the shortening is appropriate. We also refer to slot types as *slots*, to plug types as *plugs*, and to extension types as *extensions*, if the distinction between type and instance emanates from the context.

In the CM, an extension meta element represents a real extension object of a .NET application in an one-to-one relation (see Fig. 5). The *Object* property of an extension references the associated .NET object. In the same way, an extension type meta element represents the according .NET type from which the .NET object was created. The *Type* property of an extension type references the associated .NET type. In this thesis, types are depicted as dashed boxes, and instances are depicted as boxes.

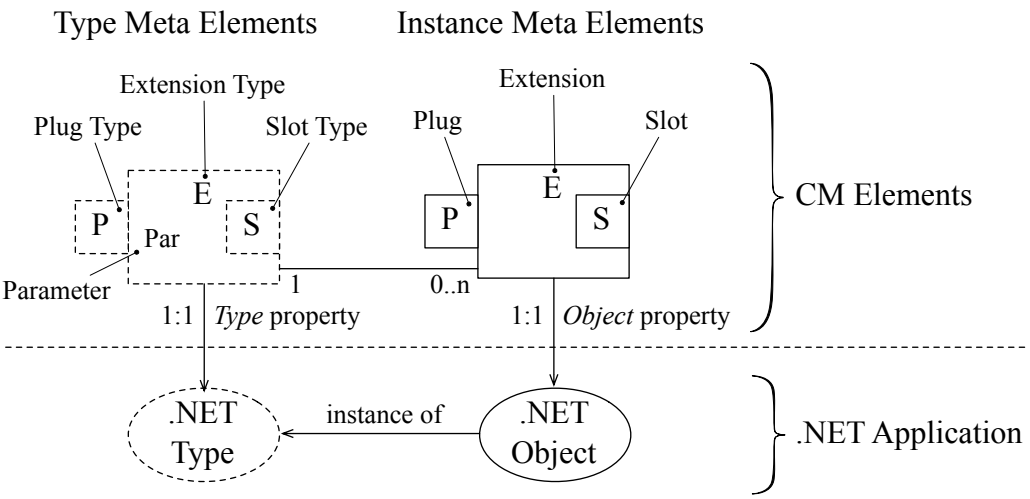


Figure 5. Relationships between meta elements and application objects

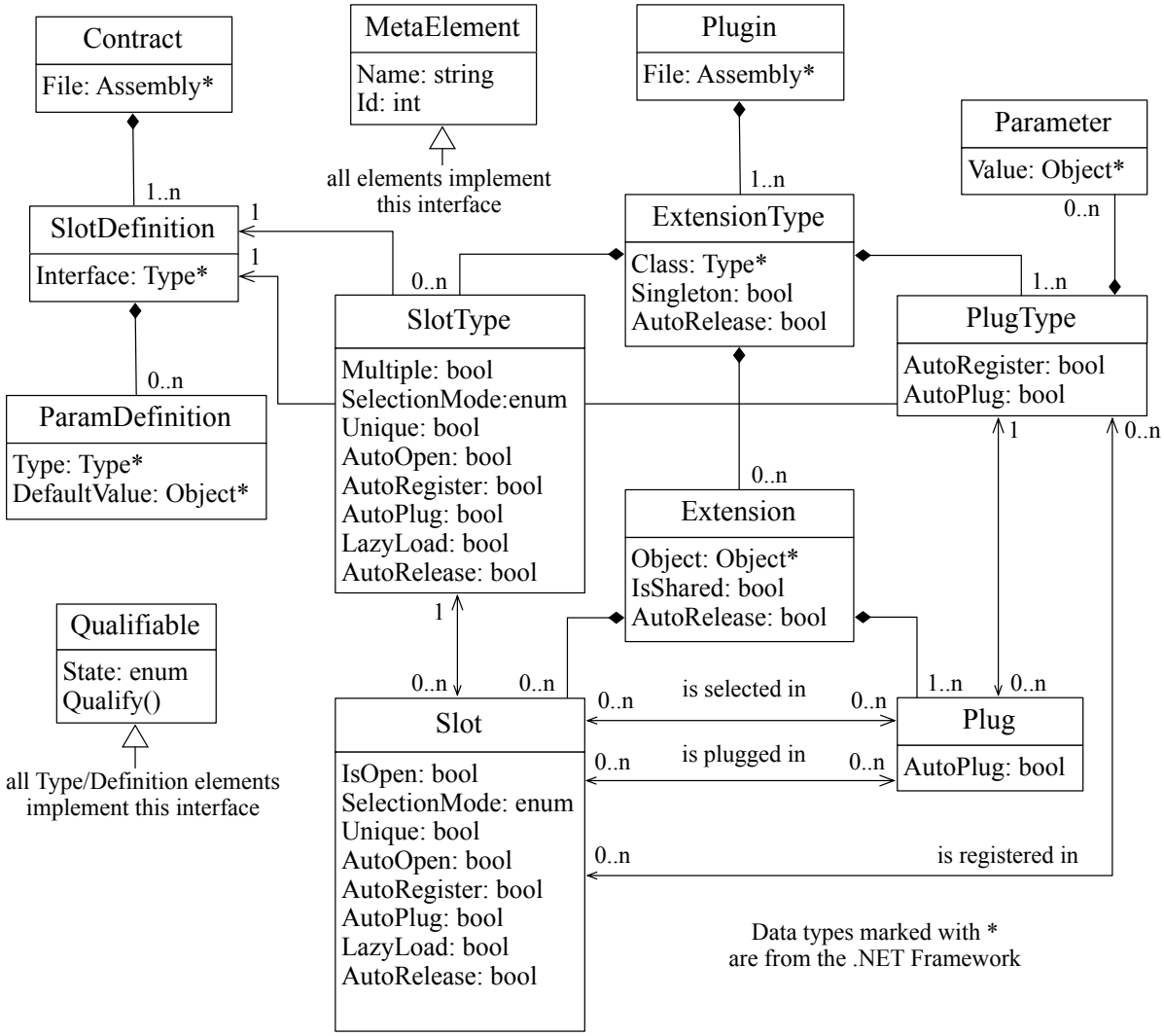


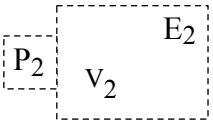
Figure 6. Class diagram of Plux.NET composition model

The class diagram in Fig. 6 shows the structure of the CM. It shows the meta elements, their attributes, and the relationships between meta elements. Every meta object can be identified by its name or by an identification number. Slot definitions must be named uniquely. Other types must be named uniquely within their parent element. Instances derive their name from their corresponding types. To distinguish instances with the same name, the composition service numbers instances consecutively per type. In the remainder of this chapter, we will show algorithms which use the attributes and relationships shown in the class diagram.

3.5 Discovering Extensions

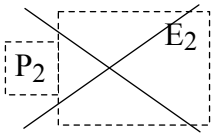
Discovery plug-ins detect when components are added to or removed from a monitored component repository and provide meta data for the components. The discovery service integrates these discovery plug-ins and provides an *Add* operation and a *Remove* operation.

A discovery plug-in calls the *Add(Contract)* operation to add slot definitions, or the *Add(Plugin)* operation to add extensions to the CM. After the discovery service has added an extension, the extension is known in the CM. In the figure below, the discovery service has added extension E_2. Extension E_2 has a plug P_2 and a parameter value V_2.



Extension E_2 has been **ADDED**
The extension is **known in the CM**.

A discovery plug-in calls the *Remove(Contract)* operation to remove slot definitions, or the *Remove(Plugin)* operation to remove extensions from the CM. After the discovery service has removed an extension, the extension is no longer known in the CM. In the figure below, the discovery service removed extension E_2.



Extension E_2 was **REMOVED**
The extension is no longer **known in the CM**.

3.6 Qualifying Extensions

Before types can be used in composition, the qualification service checks whether a type qualifies. If a type does not qualify, it is ignored in composition. The *Qualify(TypeMetaElement)* operation qualifies a type, if it complies with the standards of the CM. Table 5 (on page 37) lists the qualification rules for types. When a rule requires a unique name, that means that among multiple types with the same name, only one can be qualified. Which of multiple types with the same name qualifies, depends on the sequence of qualification. For example, the CM could contain multiple slot definitions with the same name. The slot definition which is tried first will qualify, any subsequent attempt to qualify another slot definition with the same name will fail, because its name is not unique.

Extensions, plugs, and parameters must comply with additional rules (bold typeface in Table 5). If an extension wants to contribute to a host extension, their slot and plug specifications have to match, i.e. they have to specify the same slot name. The name specifies the agreed

slot definition. A plug qualifies for a slot, if the implementation class in the extension provides a parameter-less constructor and an implementation for the interface in the slot definition. If the slot definition has parameters, the plug must provide qualified parameter. A parameter qualifies, if its type is assignable to the data type in the parameter definition. If a plug does not qualify, the composition service issues a warning and ignores the plug.

The qualification states of types can change, when types are dynamically added or removed. For example, if a plug specifies a slot definition which is not available, it does not qualify. When the according slot definition is added later, the plug is checked again. Vice versa, when a slot definition is removed, affected slots and plugs have to be checked again.

Extensions are qualified in a tolerant way. An extension with multiple plugs qualifies, if at least one plug qualifies. Other plugs which do not qualify yet, may qualify later, when new contracts are discovered.

Type meta element	qualifies if
Parameter definition	name is unique within slot definition.
Slot definition	name is unique.
Contract	name is unique within model, and contains at least one slot definition.
Parameter value	name is unique within plug type, and a parameter definition with this name exists in slot definition, and **value is assignable to parameter definition in slot definition**.
Slot type	name is unique within extension type, and a slot definition with this name exists, and slot definition qualifies.
Plug type	name is unique within extension type, and a slot definition with this name exists, and slot definition qualifies, and extension type qualifies, and **implementation class in extension type implements interface in slot definition,** and **parameter values qualify**.
Extension type	name is unique within plug-in, and at least one plug type qualifies, and **implementation class provides a parameter-less constructor**.
Plug-in	name is unique within model, and contains at least one extension type.

Table 5. Qualification rules for type meta elements

3.7 Composing Extensions

The composition service composes an application by creating extensions and by maintaining relationships between them. Relationships between host and contributor define their integration.

3.7.1 Relationships between Extensions

The CM defines two levels of host-contributor integration. Type integration integrates contributor types by retrieving their parameter values. Instance integration means that a host instantiates contributors and calls methods from their provided interfaces.

The CM defines three relationships for host-contributor integration:

- *Registered*. The *registered* relationship means type integration and connects a slot with a plug type in the CM. A plug of a contributor can be registered in the slot of a host, if slot and plug match and if the plug qualifies. After a contributor has been registered in a host, it is known to the host but not yet instantiated.
- *Plugged*. The *plugged* relationship means instance integration and connects a slot with a plug in the CM. It requires a previous *registered* relationship. A plug of a contributor can be plugged into the slot of a host. After a contributor has been plugged into a host, the contributor is in use.
- *Selected*. The *selected* relationship selects a plugged relationship. It requires a previous *plugged* relationship. After a contributor has been selected, the contributor has the focus. The selection can apply to one or several plugs connecting to a slot, depending on how the slot has been configured.

Fig. 7a shows the meta objects required for host-contributor integration and their relationships in the CM. In the remainder of this thesis we use the compact representation shown in Fig. 7b.

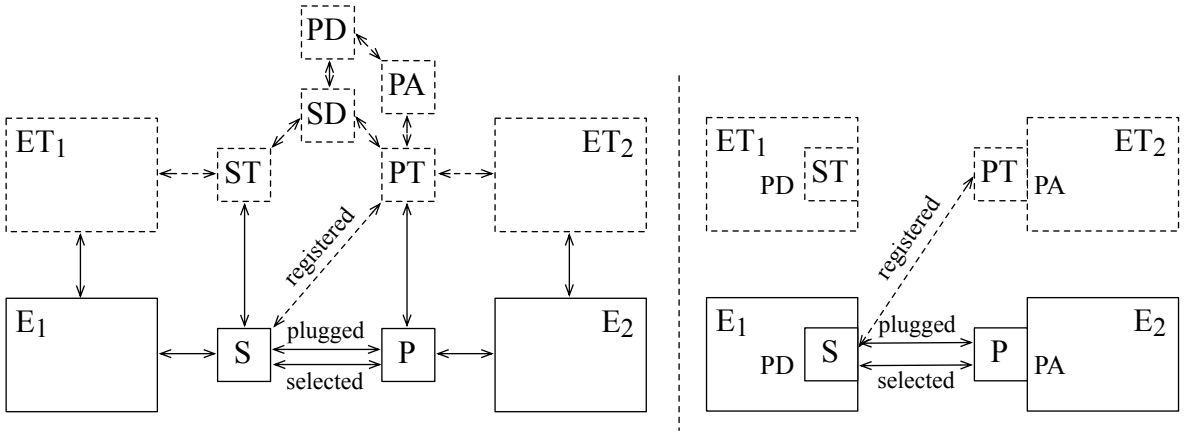


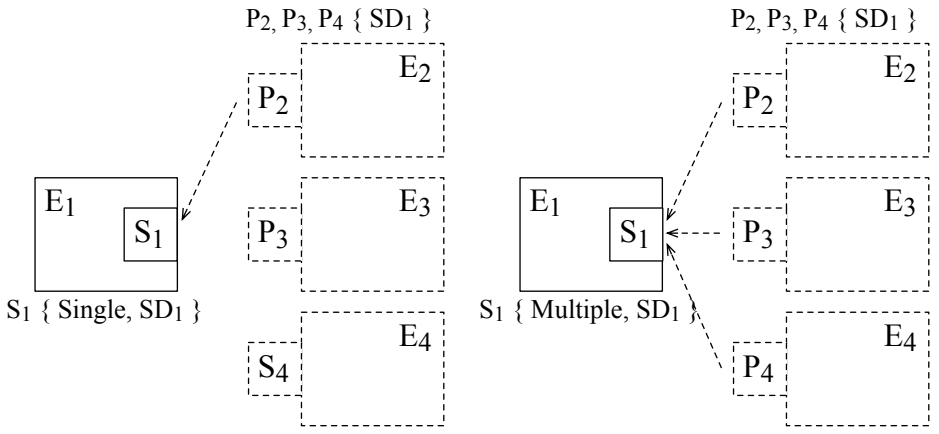
Figure 7. Relationships between meta elements in host and contributor

To *register* an extensions means to make its type and its metadata known to a host without instantiating the extension. To *plug* an extension means to create an instance of it and to connect it with the host. In most cases, registration of an extension is immediately followed by plugging. In certain situations, however, plugging should be delayed until the extension is actually used. Delayed plugging corresponds to *lazy loading* of extensions which helps to shorten application startup. For example, an application might use a menu for selecting a number of actions that are implemented as separate extensions. Registering those extensions makes their metadata (e.g., their parameters) known. Every menu command extension provides a menu command string as a parameter, which is used by the host to build and display the menu. If the user selects a command from the menu the host plugs the corresponding extension, i.e. the extension would be instantiated and connected to the host. The host now calls methods of the extension in order to perform the desired action.

To *select* an extension means to put the focus on a plugged contributor. For example, an extension might use a menu where each entry represents one of multiple child windows in a multiple document application. The window host displays the child windows, while the menu host manages the menu. The agreed collaboration protocol specifies that all plugged contributors represent visible child windows, and that the selected child window has the focus. The menu host displays a menu item for each plugged contributor in the window host. After the user selects a command from the menu, the menu host selects the corresponding contributor in the slot of the window host. The window host reacts to the selection by putting a focus on the child window associated with the selected contributor.

Register Cardinality

Slots can configure the cardinality for registered plugs. A slot with cardinality *single* can register one plug (see Fig. 8a), whereas a slot with cardinality *multiple* can register multiple plugs (see Fig. 8b). Cardinality specifies the maximum number of registered plugs, a minimum number cannot be specified. In single cardinality slots, the composition service registers the plug which was discovered first. The property *Multiple* controls wether a slot allows multiple contributors (see class *Slot Type* in Fig. 6 on page 35). The default setting is *false*.



a) Register a single Plug in a Slot b) Register multiple Plugs in a Slot

Figure 8. Slots with single or multiple cardinality

3.7.2 Creating Extensions

Extensions can be unique or shared. A unique extension can only connect to a single slot, whereas a shared extension can be plugged into several slots. The property *Unique* controls whether a slot requires unique or shared contributors (see classes *Slot* and *Slot Type* in Fig. 6 on page 35). The default setting is shared.

Contributor extensions can be singletons. A singleton extension can only have on instance. The property *Singleton* controls whether an extension is a singleton (see class *Extension Type* in Fig. 6 on page 35). The default setting is *false*.

The composition service provides operations for creating shared or unique extensions, and operations for releasing an extension. The *Create* operations create an instance meta element and the actual .NET object from the .NET class which implements the functional component.

In the remainder of this chapter, we use pseudo-code to show how the composition operations work. Italic typeface highlights variables and references to other operations.

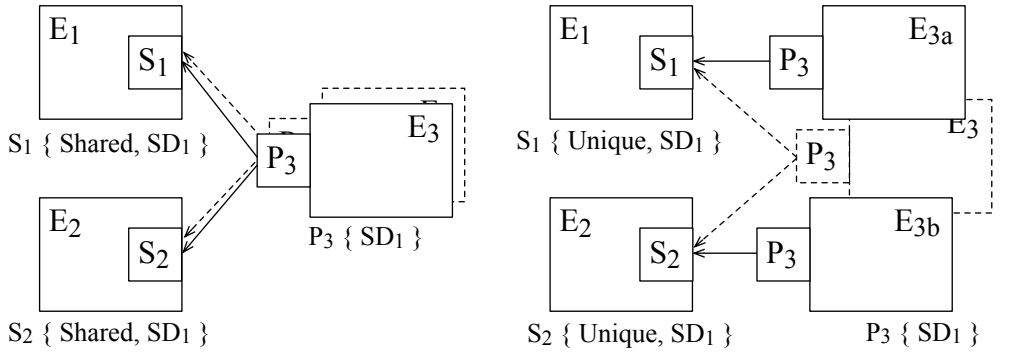
- The *CreateSharedExtension(ExtensionType)* operation creates one extension per extension type which is designated as the shared extension. The shared extension instance is shared among slots.

```
CreateSharedExtension(ExtensionType)
    if(not Qualify(ExtensionType)) return
    if(ExtensionType.IsSingleton and instance exists) return
    E = create extension meta object for ExtensionType
    make E shared instance of ExtensionType
    E.Object = create .NET object for ExtensionType.Class
    // Lazy Loading
    for(all plugs P of extension E)
        PT = type of P
        for(all slots S where PT is registered)
            if(S.LazyLoad and S.AutoPlug and not S.Unique)
                Plug(S, P)
    return E
```

The last section of the pseudo code implements lazy loading for shared slots. If a shared contributor is created, it is plugged into all shared lazy-loading slots where the contributor is registered.

In Fig. 9a, hosts E_1 and E_2 have opened slots S_1 and S_2. Both slots use the same slot definition SD_1, are configured for shared contributors, and have plug P_3 of contributor E_3 registered. The composition service has created a shared extension E_3 and has plugged it into both slots S_1 and S_2.

In this thesis, the *registered* relationship is depicted with a dashed line with an arrow on the slot side. Extension types are depicted as boxes with dashed lines. The *plugged* relationship is depicted with a solid line with an arrow on the slot side. Extensions are depicted as solid boxes.



a) Share Contributor among Slots b) Use unique Contributors exclusively in Slot

Figure 9. Slots with shared or unqiue contributors

- The *CreateUniqueExtension(ExtensionType)* operation creates a unique extension of an extension type. Per extension type, an arbitrary number of extensions can be created. For each slot which is configured for unique contributors, the composition service creates a unique contributor.

```
CreateUniqueExtension(ExtensionType)
    if(Not Qualify(ExtensionType)) return
    if(ExtensionType.IsSingleton and instance exists) return
    E = create extension meta object for ExtensionType
    E.Object = create .NET object for ExtensionType.Class
    return E
```

  In Fig. 9b, hosts E_1 and E_2 have opened slots S_1 and S_2. Both slots use the same slot definition SD_1, are configured for unique contributors, and have plug P_3 of contributor E_3 registered. The composition service has created unique extensions E_{3a} and E_{3b}, and has plugged them into slot S_1 and S_2 respectively.

- The *Release(Extension)* operation closes all slots of the extension, disposes the .NET object and the instance meta object. Before the slots are closed, the extension is unplugged from all hosts where it is plugged.

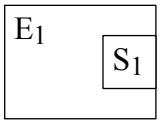
```
Release(Extension)
    UnplugWherePlugged(Extension)
    for(all slots S of Extension)
        CloseSlot(S)
    dispose .NET object
    dispose instance meta object
```

3.7.3 Maintaining Composition Relationships

The composition service provides operations for maintaining relationships between extensions. Composition operations open slots, register plugs, plug plugs, and select plugs. Decomposition operations close slots, deselect plugs, unplug plugs, and deregister plugs. The operations in detail are:

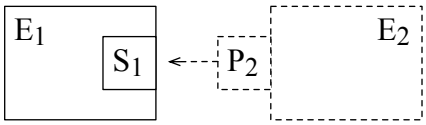
- The *OpenSlot(Slot)* operation opens a slot. After a slot has been opened, it is registering contributors. In the figure below, the composition service opened slot S_1.

  OpenSlot(Slot)
  if(*Slot.IsOpen*) return
  change state of *Slot* to opened
  RegisterAll(Slot)



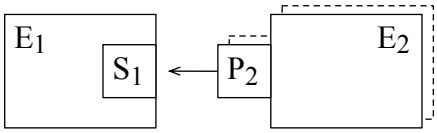
Slot S_1 was **OPENED**
The host is **registering** contributors.

- The *Register(Slot, PlugType)* operation registers a plug type of a contributor in a slot of a host. After a contributor has been registered, the contributor's type is known. In the next step, the composition service will create and plug registered contributors (see more operations and pseudo-code on page 44). In the figure below, plug P_2 has been registered in slot S_1 and thus host E_1 has integrated the type of contributor E_2.



Plug type P_2 was **REGISTERED** in slot S_1
The contributor is **known** to the host.

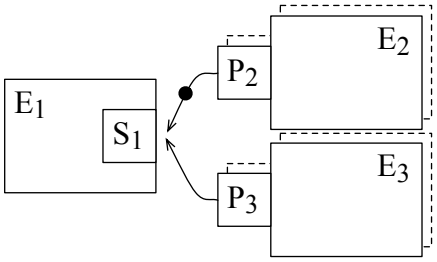
- The *Plug(Slot, Plug)* operation plugs a plug of a registered contributor into a slot of a host. After a contributor has been plugged, the contributor is in use. In the next step, the composition service will open the slots of the contributor (see more operations and pseudo-code on page 46). In the figure below, plug P_2 has been plugged into slot S_1 and thus host E_1 has integrated the instance of contributor E_2.



Plug P_2 was **PLUGGED** in slot S_1
The contributor is **in use**.

- The *Select(Slot, Plug)* operation selects a plug of a plugged contributor. After a contributor has been selected, the contributor has the focus. In the figure below, plug P_2 has been selected, and thus host E_1 has set the focus on contributor E_2. In this thesis, the selection is depicted with a black circle on the line which depicts the *plugged* relationship.

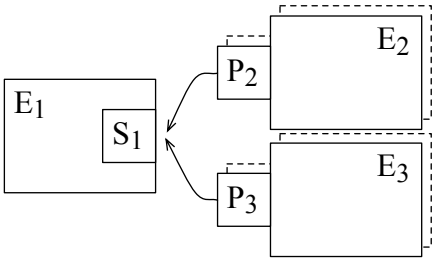
  Select(Slot, Plug)
  if(*Plug* is selected in *Slot*) return
  if(*Plug* is not plugged in *Slot*) return
  add selected relationship *Slot/Plug*



Plug P_2 was **SELECTED** in slot S_1
The contributor **has the focus**.

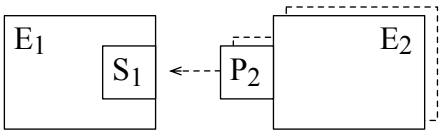
- The *Deselect(Slot, Plug)* operation deselects a plug of a selected contributor. After a contributor has been deselected, the contributor looses the focus. In the figure below, plug P_2 has been deselected, and thus host E_1 has removed the focus from contributor E_2.

  Deselect(Slot, Plug)
  if(*Plug* is not selected in *Slot*) return
  if(*Plug* is not plugged in *Slot*) return
  remove selected relationship *Slot/Plug*



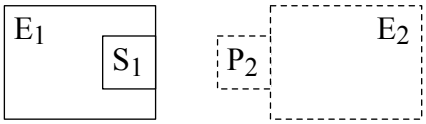
Plug P_2 was **DESELECTED** in slot S_1
The contributor **lost the focus**.

- The *Unplug(Slot, Plug)* operation unplugs a plug of a plugged contributor from a slot of a host. After a contributor has been unplugged, the contributor goes out of use (see more operations and pseudo-code on page 49). In the figure below, plug P_2 has been unplugged from slot S_1, and thus host E_1 stops using contributor E_2. If plug P_2 was selected, it is deselected before it is unplugged. If contributor E_2 is not plugged in any other host, it is released. Plug P_2 stays registered.



Plug P_2 was **UNPLUGGED** in slot S_1
The contributor goes **out of use**.

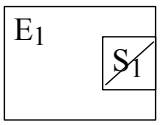
- The *Deregister(Slot, PlugType)* operation deregisters a plug type of a registered contributor from a slot of a host. After a contributor has been deregistered, the contributor's type is no longer known to the host (see more operations and pseudo-code on page 45). In the figure below, plug P_2 has been deregistered from slot S_1., and thus host E_1 has disintegrated contributor E_2. If contributors of type E_2 were plugged into slot S_1, they are unplugged before the contributor is deregistered.



Plug type P_2 was **DEREGISTERED** in slot S_1
The contributor is **no longer known** to the host.

- The *CloseSlot(Slot)* operation closes a slot. After a slot was closed, it is deregistering contributors. In the figure below, the composition service closed slot S_1. In this thesis, a closed slot is depicted strike-through.

  CloseSlot(Slot)
  if(not *Slot.IsOpen*) return
  change state of *Slot* to closed
  DeregisterAll(Slot)



Slot S_1 was **CLOSED**
The host is **deregistering** contributors.

Registering Contributors

The composition service provides multiple operations for registering contributors in hosts. Fig. 10 overviews operations which register individual plugs in individual slots (a-c), and operations which register all plugs of contributors in all matching slots of hosts (d-f):

a) *Register(Slot, PlugType)* registers a matching and qualified plug type in a slot. In the next step, the *CreateAndPlug* operation creates and plugs the contributor (see page 46).

```
Register(Slot, PlugType)
    if(not Slot.IsOpen) return
    if(PlugType does not match Slot) return
    if(PlugType is registered in Slot) return
    if(not Slot.Multiple and a plug type is registered in Slot) return
    if(not Qualified(PlugType)) return
    add registered relationship Slot/PlugType
    CreateAndPlug(Slot, PlugType)
```

b) *RegisterPlug(PlugType)* registers a plug type in all matching slots.

```
RegisterPlug(PlugType)
    for(all slots S matching PlugType)
        Register(S, PlugType)
```

c) *RegisterAll(Slot)* registers all matching plugs in a slot.

```
RegisterAll(Slot)
    if(Slot.Multiple)
        for(all plug types PT)
            Register(Slot, PT)
    else
        PT = first discovered plug type matching Slot
        Register(Slot, PT)
```

d) *Register(Extension, ExtensionType)* registers a contributor extension type in a host extension.

```
Register(Extension, ExtensionType)
    for(all slots S of Extension)
        for(all plug types PT of ExtensionType)
            Register(S, PT)
```

e) *RegisterPlugs(ExtensionType)* registers a contributor extension type in all hosts.

```
RegisterPlugs(ExtensionType)
    for(all plug types PT of ExtensionType)
        RegisterPlug(PT)
```

f) *RegisterAll(Extension)* registers all matching contributors in a host extension.

```
RegisterAll(Extension)
    for(all slots S of Extension)
        RegisterAll(S)
```

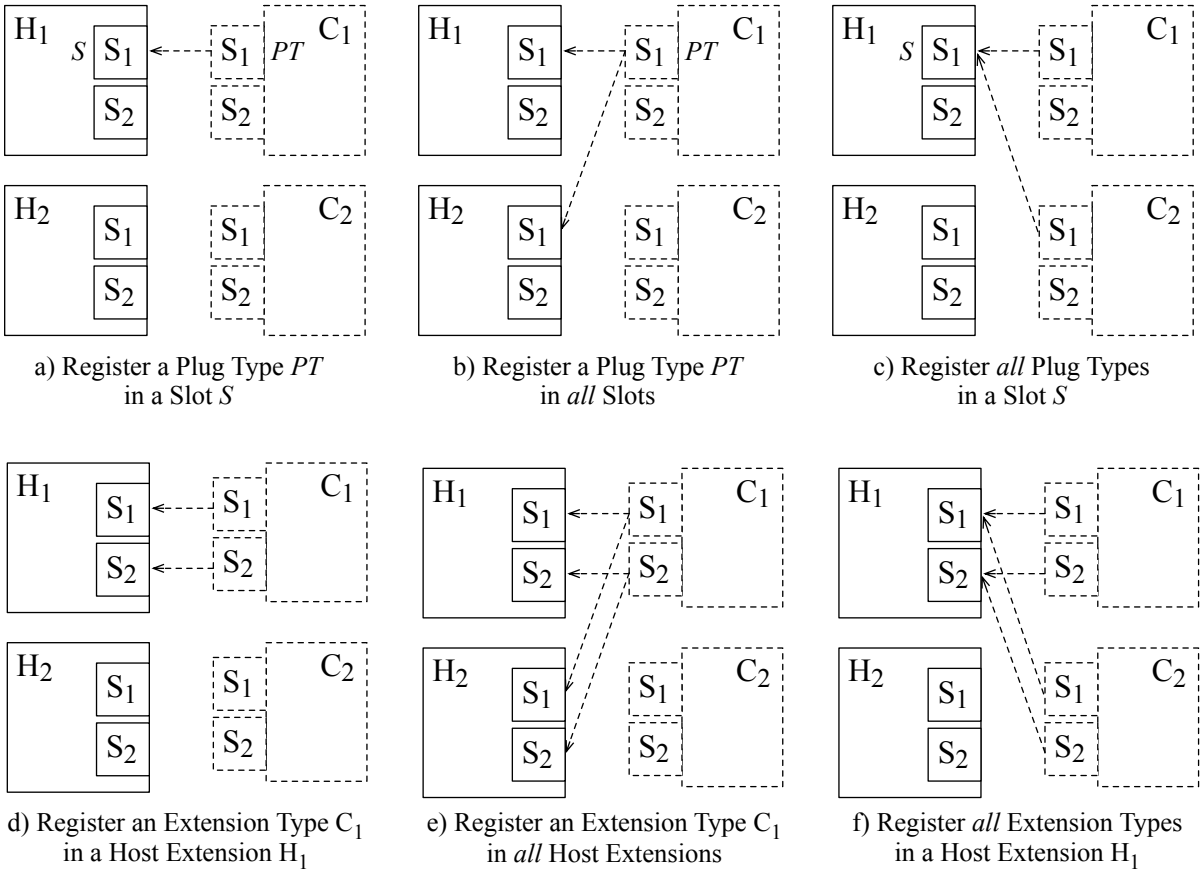


Figure 10. Composition operations for registration

Deregistering Contributors

The composition service provides multiple operations for deregistering contributors from hosts.

a) *Deregister(Slot, PlugType)* deregisters a registered plug type from a slot.

```
Deregister(Slot, PlugType)
    if(PlugType is not registered in Slot) return
    for(all plugs P connected to Slot)
        Unplug(Slot, P)
    remove registered relationship Slot/PlugType
```

b) *DeregisterPlug(PlugType)* deregisters a plug type from all slots where it is registered.

```
DeregisterPlug(PlugType)
    for(all slots S in which PlugType is registered)
        Deregister(S, PlugType)
```

c) *DeregisterAll(Slot)* deregisters all plugs from a slot.

```
DeregisterAll(Slot)
    for(all plug types PT which are registered in Slot)
        Deregister(Slot, PT)
```

d) *Deregister(Extension, ExtensionType)* deregisters a registered contributor from a host.

```
Deregister(Extension, ExtensionType)
    for(all slots S of Extension)
        for(all plug types PT of ExtensionType)
            Deregister(S, PT)
```

e) *DeregisterPlugs(ExtensionType)* deregisters a contributor from all hosts where it is registered.

```
DeregisterPlugs(ExtensionType)
    for(all plug types PT of ExtensionType)
        DeregisterPlug(PT)
```

f) *DeregisterAll(Extension)* deregisters all contributors from a host.

```
DeregisterAll(Extension)
    for(all slots S of Extension)
        DeregisterAll(S)
```

Plugging Contributors

The composition service provides multiple operations for plugging contributors into hosts. Fig. 11 overviews the operations which plug individual plugs in individual slots (a-c), and operations which plug all plugs of contributors in all matching slots of hosts (d-f):

a) *Plug(Slot, Plug)* plugs a plug into a slot. In the next step, the *OpenSlot* operation opens the slots of the plugged contributor.

```
Plug(Slot, Plug)
    if(Plug is plugged in Slot) return
    PlugType = type of Plug
    if(PlugType is not registered in Slot) return
    add plugged relationship Slot/Plug
    Select(Slot, Plug)
    E = extension containing Plug
    for(all slots S of extension E)
        OpenSlot(S)
```

CreateAndPlug(Slot, PlugType) combines the create and the plug operation. It creates a unique or shared contributor, depending on the configuration of the slot, and calls the *Plug* operation. If the contributor is a singleton and one instance already exists, a subsequent attempt to create a new instance fails. This can happen when the singleton instance is shared, and a slot tries to create a unique contributor, or vice versa.

```
CreateAndPlug(Slot, PlugType)
    if(PlugType is not registered in Slot) return
    ExtensionType = extension containing PlugType
    if(Slot.Unique)
        Contributor = CreateUniqueExtension(ExtensionType)
    else
        if(shared extension of ExtensionType exists)
            Contributor = shared extension of ExtensionType
```

```
        else
            Contributor = CreateSharedExtension(ExtensionType)
    if(Contributor == null) return // singleton existed
    Plug = plug of Contributor which matches Slot
    Plug(Slot, Plug)
```

b) *PlugWhereRegistered(Plug)* plugs a plug into all slots where it is registered.

```
PlugWhereRegistered(Plug)
    PlugType = type of Plug
    for(all slots S where PlugType is registered)
        Plug(S, Plug)
```

CreateAndPlugWhereRegistered(PlugType) combines the create and the plug operation. The result depends on the configuration of the slot. In shared slots, the contributor is shared amongst slots (see b1 in Fig. 11). In unique slots, a unique contributor is created for each slot (see b2 in Fig. 11).

```
CreateAndPlugWhereRegistered(PlugType)
    for(all slots S where PlugType is registered)
        CreateAndPlug(S, PlugType)
```

c) *CreateAndPlugAllRegistered(Slot)* creates and plugs all contributors which are registered in a slot.

```
CreateAndPlugAllRegistered(Slot)
    for(all plug types PT which are registered in Slot)
        CreateAndPlug(Slot, PT)
```

d) *Plug(*$Extension_{Host}$, $Extension_{Contributor}$*)* plugs a contributor into a host.

```
Plug(Extension_Host, Extension_Contributor)
    for(all slots S of Extension_Host)
        for(all plugs P of Extension_Contributor)
            Plug(S, P)
```

CreateAndPlug(Extension, ExtensionType) combines the create and the plug operation.

```
CreateAndPlug(Extension, ExtensionType)
    for(all slots S of Extension)
        for(all plug types PT of ExtensionType)
            CreateAndPlug(S, PT)
```

e) *PlugWhereRegistered(Extension)* plugs a contributor into all hosts where it is registered.

```
PlugWhereRegistered(Extension)
    for(all plugs P of Extension)
        PlugWhereRegistered(P)
```

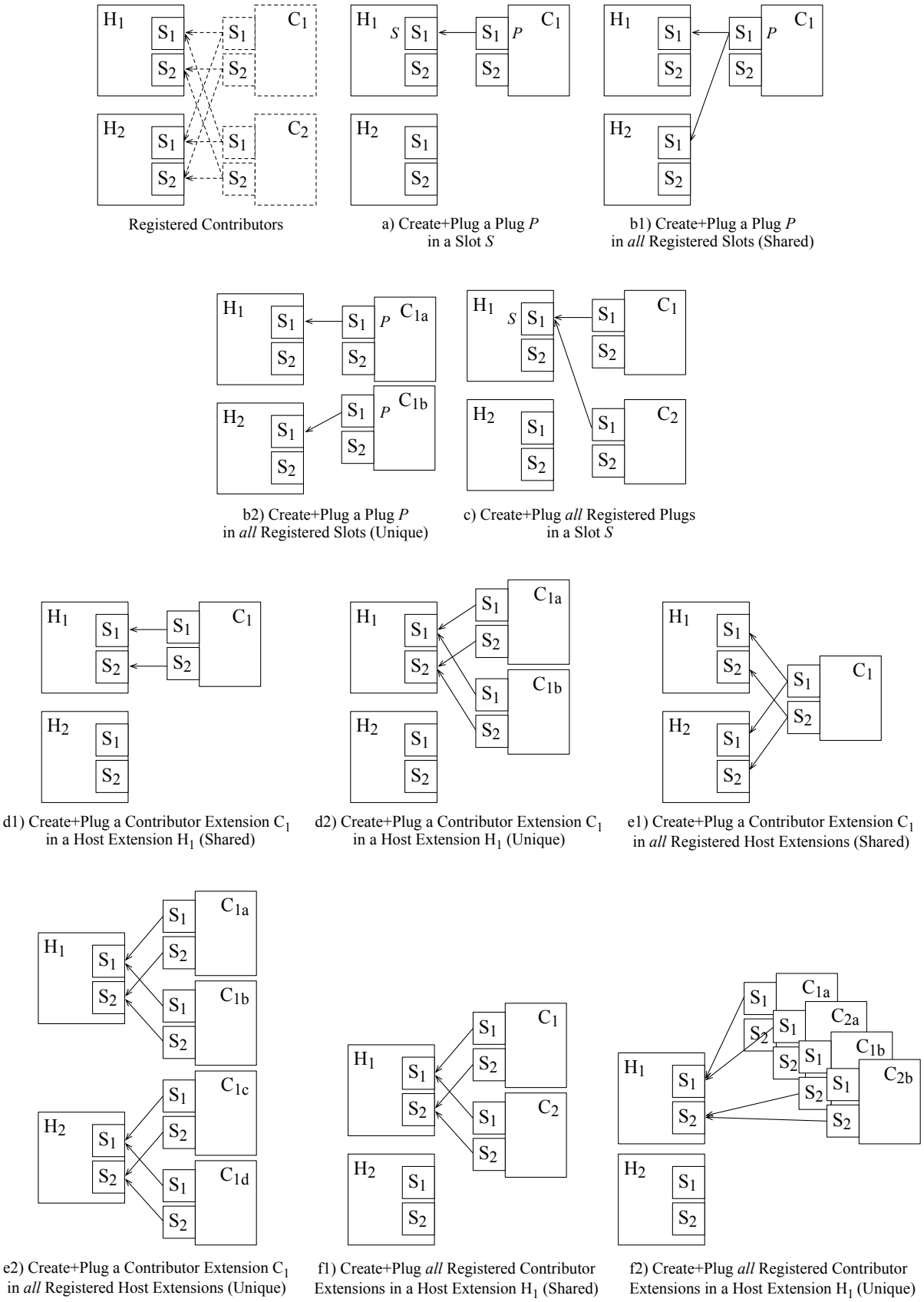


Figure 11. Composition operations for creation and plugging

CreateAndPlugWhereRegistered(ExtensionType) combines the create and the plug operation. The result depends on the configuration of the slots. In shared slots, the contributor is shared amongst slots (see d1, e1, f1 in Fig. 11). In unique slots, a unique contributor is created for each slot (see d2, e2, f2 in Fig. 11).

```
CreateAndPlugWhereRegistered(ExtensionType)
    for(all plug types PT of ExtensionType)
        CreateAndPlugWhereRegistered(PT)
```

f) *CreateAndPlugAllRegistered(Extension)* creates and plugs all contributors which are registered in a host.

```
CreateAndPlugAllRegistered(Extension)
    for(all slots S of Extension)
        CreateAndPlugAllRegistered(S)
```

Unplugging Contributors

The composition service provides multiple operations for unplugging contributors from hosts.

a) *Unplug(Slot, Plug)* unplugs a plugged plug from a slot.

```
Unplug(Slot, Plug)
    if(Plug is not plugged in Slot) return
    if(Plug is selected in Slot)
        Deselect(Slot, Plug)
    remove plugged relationship Slot/Plug
    Extension = extension containing Plug
    if(Extension is not plugged in other hosts)
        Release(Extension)
```

b) *UnplugWherePlugged(Plug)* unplugs a plug from all slots where it is plugged.

```
UnplugWherePlugged(Plug)
    for(all slots S which Plug is connected to)
        Unplug(S, Plug)
```

c) *UnplugAll(Slot)* unplugs all plugs from a slot.

```
UnplugAll(Slot)
    for(all plugs P connected to Slot)
        Unplug(Slot, P)
```

d) *Unplug($Extension_{Host}$, $Extension_{Contributor}$)* unplugs a plugged contributor extension from a host extension.

```
Unplug(Extension_Host, Extension_Contributor)
    for(all slots S of Extension_Host)
        for(all plugs P of Extension_Contributor connected to S)
            Unplug(S, P)
```

e) *UnplugWherePlugged(Extension)* unplugs a contributor extension from all host extensions where it is plugged.

UnplugWherePlugged(Extension)
for(all plugs *P* of *Extension*)
UnplugWherePlugged(P)

f) *UnplugAll(Extension)* unplugs all contributor extensions from a host extension.

UnplugAll(Extension)
for(all slots *S* of *Extension*)
UnplugAll(S)

3.7.4 Configuring Composition

To configure composition means to control through settings how the automatic procedure composes extensions. With the default settings the composition is in automatic mode. In automatic mode, the composition service opens every slot, and registers and plugs every contributor. In most scenarios this is the desired behavior. In certain situations, however, host extensions want to control manually which slots open, which contributors register, or which contributors plug. Section 5.2 (see page 95) shows scenarios for the manual mode.

Fig. 12 shows the properties which control the composition procedure. The properties can be set on different scopes: for a slot, for a plug, for an extension, or for the composition service. The connection between a settings on type level and a setting on instance level, for example slot type and slot, is such, that on instantiation the settings are copied from the type to the instance. After instantiation the settings of slot type and slot can be set independently. When the composition service evaluates a setting, the setting must be *true* on each level, in order that a composition step is performed.

In order to explain the affect of the settings, we extend the pseudo-code operations from Section 3.7.3 (see page 41). In the pseudo-code, added statements are highlighted in bold typeface:

- *AutoOpen* controls whether the composition service opens slots when a contributor is plugged. If *AutoOpen=true* for the composition service, and for a slot *S* of an extension *E,* the composition service automatically opens *S* when *E* is plugged. The property *AutoOpen* affects the *Plug(Slot, Plug)* operation.

  Plug(Slot, Plug)
  if(*Plug* is plugged in *Slot*) return
  PlugType = type of *Plug*
  if(*PlugType* is not registered in *Slot)* return
  add plugged relationship *Slot/Plug*
  E = extension containing *Plug*
  if(*CompositionService.AutoOpen*)
  for(all slots *S* of extension *E*)
  if(*S.AutoOpen*)
  OpenSlot(S)

- *AutoRegister* controls whether the composition service registers contributors when a slot is opened. If *AutoRegister=true* for the composition service, for a slot *S* of a host *H*, and for a plug type *PT* of a contributor *C*, the composition service automatically

registers *PT* in *S* when *S* opens. The property *AutoRegister* affects the operations *OpenSlot(Slot)* and *RegisterAll(Slot)*.

```
OpenSlot(Slot)
    if(Slot.IsOpen) return
    change state of Slot to opened
    if(CompositionService.AutoRegister and Slot.AutoRegister)
        RegisterAll(Slot)

RegisterAll(Slot)
    if(Slot.Multiple)
        for(all plug types PT)
            if(PT.AutoRegister)
                Register(Slot, PT)
    else
        PT = first discovered plug type matching Slot
        if(PT.AutoRegister)
            Register(Slot, PT)
```

Setting	Scope: Slot Type	Slot	Plug Type	Plug	Extension Type	Extension	Composition Service
AutoOpen	✕	✕					✕
AutoRegister	✕	✕	✕				✕
AutoPlug	✕	✕	✕	✕			✕
AutoSelect	✕	✕	✕	✕			✕
LazyLoad	✕	✕					✕
AutoRelease	✕	✕			✕	✕	✕

Figure 12. Settings for composition configuration

- *AutoPlug* controls whether the composition service creates and plugs a contributor when it is registered. If *AutoPlug=true* for the composition service, for a slot *S*, and for plug type *PT* of a contributor *C*, the composition services automatically creates *C* and plugs it into slot *S*. The property *AutoPlug* affects the operation *Register(Slot, PlugType)*.

```
Register(Slot, PlugType)
    if(not Slot.IsOpen) return
    if(PlugType does not match Slot) return
    if(PlugType is registered in Slot) return
```

if(not *Slot.Multiple* and a plug type is registered in *Slot*) return
if(not *Qualify(PlugType)*) return
if(registration creates cycle) return
add registered relationship *Slot/PlugType*
if(*CompositionService.AutoPlug* and *Slot.AutoPlug* and not Slot.LazyLoad
and *PlugType.AutoPlug*)
CreateAndPlug(Slot, PlugType)

- *AutoSelect* controls whether the composition service selects a plugged relationship when a contributor is plugged. If *AutoSelect=true* for the composition service, for a slot *S*, and for a plug *P*, the composition services automatically selects the plugged relationship of plug *P* and slot *S*. The property *AutoSelect* affects the operation *Plug(Slot, Plug)*.

Plug(Slot, Plug)
if(*Plug* is plugged in *Slot*) return
PlugType = type of *Plug*
if(*PlugType* is not registered in *Slot)* return
add plugged relationship *Slot/Plug*
if(*CompositionService.AutoSelect* and *Slot.AutoSelect* and *Plug.AutoSelect*)
Select(Slot, Plug)
E = extension containing *Plug*
for(all slots *S* of extension *E*)
OpenSlot(S)

- *LazyLoad* controls whether the composition service creates and plugs a contributor when it is registered, in a slightly different way than the *AutoPlug* property does. Both settings have in common, that when a hosts sets *AutoPlug=false* or *LazyLoad=true*, the composition service will not create and plug the contributor after registration. The host itself is responsible for that. The first difference is, that with lazy loading, the composition service will automatically plug a contributor when it is created. Whereas with auto plugging disabled, the host must manually call the plug operation after creation. The second difference matters when multiple shared slots with lazy loading have the same contributor registered. As soon as any of the slots creates the shared contributor, it is automatically plugged in all slots where the contributor is registered (see *CreateSharedExtension* operation on page 40).

- *AutoRelease* controls whether the composition service releases a contributor when it is unplugged. If *AutoRelease=true* for the composition service and for a contributor *C*, the composition service will release *C* as soon as it is unplugged from a slot S that also has *AutoRelease=true*, but only if *C* is not plugged in any other slots.

Unplug(Slot, Plug)
if(*Plug* is not plugged in *Slot*) return
if(Plug is selected in Slot)
Deselect(Slot, Plug)
remove plugged relationship *Slot/Plug*
Contributor = extension containing *Plug*
if(*CompositionService.AutoRelease* and *Slot.AutoRelease*

and *Contributor.AutoRelease*
if(*Contributor* is not plugged in other hosts)
Release(Contributor)

3.8 Extension Life-Cycle

Discovery and qualification operations change the state of types, the creation operations create and dispose instances, and composition operations change the state of types and instances. The allowed sequence of these operations specify a life-cycle for types and instances.

3.8.1 Type Life-Cycle

The life-cycle of a type begins as soon as it is discovered. The state diagram in Fig. 13 shows that the *Add* operation changes the state of a type to *discovered*. The life-cycle of a type ends when the *Remove* operation disposes the type. The *Qualify* operation changes the state of a qualifiable type to *qualified*. If qualification fails, the state of the type remains *discovered*. The state *qualified* can be reset to *discovered*, when other types to which a type has dependencies are removed.

The *registered* state applies to plug types only. The *Register* operation of the composition service registers a qualified plug in a slot and changes the state of the plug to *registered*. The *registered* state of a plug applies in the context of a slot, because a plug can be registered in multiple slots. The *Deregister* operation deregisters a plug from a slot and changes the state of the plug back to *qualified*. The *Remove* operation automatically deregisters a *registered* type.

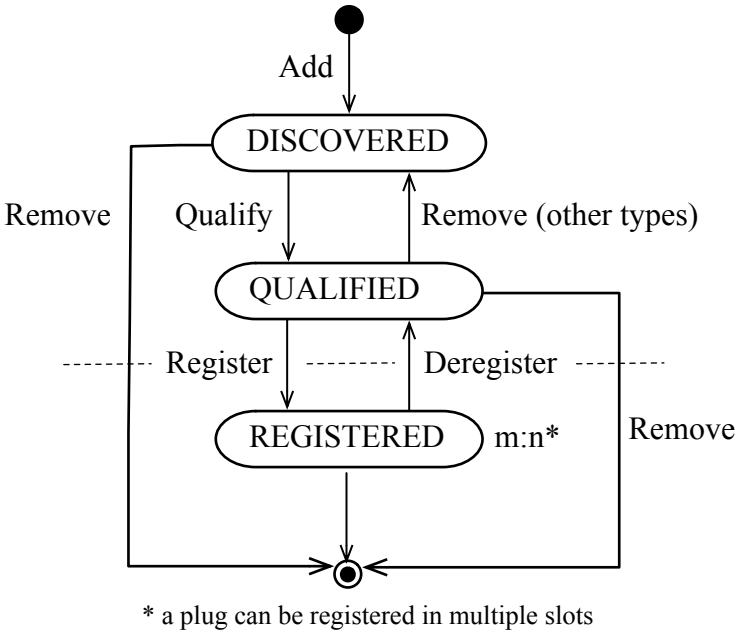
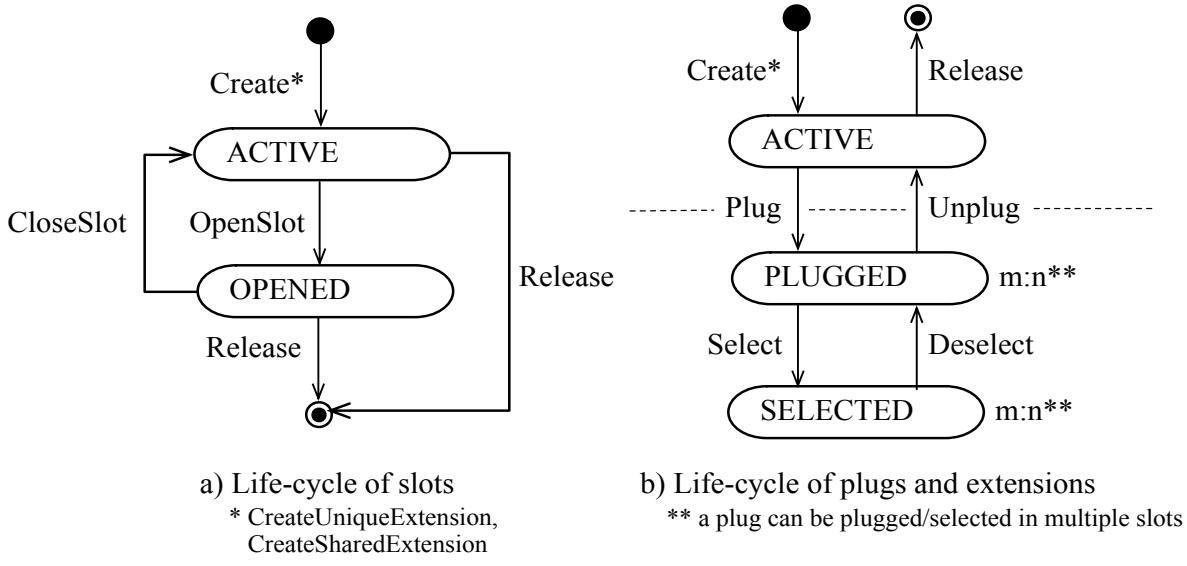


Figure 13. State diagram of type meta element life-cycle

3.8.2 Instance Life-Cycle

The life-cycle of an instance begins when it is created by the composition service. The state diagrams in Fig. 14a+b show that the *CreateUniqueExtension* operation or the *CreateShared-Extension* operation changes the state of an extension to *active*. The life-cycle of a slot/extension ends when the *Release* operation disposes the slot/extension.

The states for slots (see Fig. 14a) differ from states of extensions and plugs (see Fig. 14b). The *OpenSlot* operation changes the state of a slot to *opened*. The *CloseSlot* operation changes the state of a slot back to *active*. The *Release* operation automatically closes an *opened* slot.



a) Life-cycle of slots
* CreateUniqueExtension, CreateSharedExtension

b) Life-cycle of plugs and extensions
** a plug can be plugged/selected in multiple slots

Figure 14. State diagram of instance meta element life-cycle

The *plugged* and *selected* states applies to plugs only. The *Plug* operation plugs an *active* plug into an *opened* slot and changes the state of the plug to *plugged*. The *plugged* state of a plug applies in the context of a slot, because a plug can be plugged in multiple slots. The *Plug* operation requires that the corresponding plug type is *registered* in the slot. The *Unplug* operation unplugs a *plugged* plug from a slot and changes the state of the plug back to *active*. The *Release* operation automatically unplugs a *plugged* plug. The *Select* operation selects a plug and changes the state of the plug to *selected*. The selected state of plug applies in the context of a slot, because a plug can be selected in multiple slots. The *deselect* operation deselects a selected plug and changes the state of the plug back to *plugged*. The *Unplug* operation automatically deselects a *selected* plug.

3.9 Composing an Application

In this section, we revisit the operations of the composition service, to add support for the event-based programming model in hosts. Then we define a small core application, which is used to start application composition. A detailed example will show how the composition ser-

vice creates the core application, and how it composes an application by dynamically adding, removing, and replacing extensions.

3.9.1 Notifying Hosts and Contributors with Events

Operations of the composition service only change the state of meta objects. They do not directly affect the state of the underlying .NET objects. For example, the plug operation connects a slot meta object of a host to the plug meta object of a contributor. To allow the .NET object of the host to communicate with the .NET object of the contributor, the composition service sends a *Plugged* event to the host. Generally, whenever the composition state of an application changes, e.g., by opening slots or by creating, plugging or unplugging extensions, the involved extensions get notified by an event to which they can react. Fig. 15 shows the composition events and which notifications are sent.

In the following, we extend the pseudo-code operations from Section 3.7.3 (see page 41) with the event notification. In the pseudo-code, added statements are highlighted in bold typeface:

- *Opened-Event*. The *OpenSlot* operation opens a slot and notifies the host to which this slot belongs.

```
OpenSlot(Slot)
    if(Slot.IsOpen) return
    change state of Slot to opened
    send Opened event to slot host
    if(CompositionService.AutoRegister and Slot.AutoRegister)
        RegisterAll(Slot)
```

- *Registered-Event*. The *Register* operation registers a plug type in a slot and notifies the host to which this slot belongs to.

```
Register(Slot, PlugType)
    if(not Slot.IsOpen) return
    if(PlugType does not match Slot) return
    if(PlugType is registered in Slot) return
    if(not Slot.Multiple and a plug type is registered in Slot) return
    if(not Qualify(PlugType)) return
    if(registration creates cycle) return
    add registered relationship Slot/PlugType
    send Registered event to slot host
    if(CompositionService.AutoPlug and Slot.AutoPlug and not Slot.LazyLoad
       and PlugType.AutoPlug)
        CreateAndPlug(Slot, PlugType)
```

- *Plugged-Event*. The *Plug* operation connects a plug of a contributor with the slot of host and notifies the contributor and the host.

```
Plug(Slot, Plug)
    if(Plug is plugged in Slot) return
    PlugType = type of Plug
    if(PlugType is not registered in Slot) return
```

add plugged relationship *Slot/Plug*
send *Plugged* event to contributor
send *Plugged* event to slot host
if(*CompositionService.AutoSelect* and *Slot.AutoSelect* and *Plug.AutoSelect*)
Select(Slot, Plug)
E = extension containing *Plug*
if(*CompositionService.AutoOpen*)
for(all slots *S* of extension *E*)
if(*S.AutoOpen*)
OpenSlot(S)

Notification sent to...

Slot S								Plug P				Ext E		
Opened	Registered	Plugged	Selected	Deselected	Unplugged	Deregistered	Closed	Plugged	Selected	Deselected	Unplugged	Created	Released	
✕														slot S was opened
	✕													a plug type was registered for slot S
												✕		contributor E was created
		✕						✕						a plug P was plugged into slot S
			✕						✕					a plug P was selected in slot S
				✕						✕				a plug P was deselected in slot S
					✕						✕			a plug P was unplugged from slot S
													✕	contributor E was released
						✕								a plug type was deregistered from slots s
							✕							slot S was closed

Composition event

Figure 15. Composition notifications for host and contributor

- *Selected-Event*. The *Select* operation selects a plug and notifies the contributor and the host.

Select(Slot, Plug)
if(*Plug* is selected in *Slot*) return
if(*Plug* is not plugged in *Slot*) return
add selected relationship *Slot/Plug*
send *Selected* event to contributor
send *Selected* event to slot host

- *Deselected-Event*. The *Deselect* operation deselects a plug and notifies the contributor and the host.

Deselect(Slot, Plug)
if(*Plug* is not selected in *Slot*) return
if(*Plug* is not plugged in *Slot*) return
remove selected relationship *Slot/Plug*

send *Deselected* event to contributor
send *Deselected* event to slot host

- *Unplugged-Event*. The *Unplug* operation unplugs a plug of a contributor from the slot of a host and notifies contributor and host.

```
Unplug(Slot, Plug)
    if(Plug is not plugged in Slot) return
    if(Plug is selected in Slot)
        Deselect(Slot, Plug)
    remove plugged relationship Slot/Plug
    send Unplugged event to contributor
    send Unplugged event to slot host
    Contributor = extension containing Plug
    if(CompositionService.AutoRelease and Slot.AutoRelease
       and Contributor.AutoRelease
        if(Contributor is not plugged in other hosts)
            Release(Contributor)
```

- *Deregistered-Event*. The *Deregister* operation deregisters a contributor plug type from the slot of a host and notifies the host.

```
Deregister(Slot, PlugType)
    if(PlugType is not registered in Slot) return
    for(all plugs P connected to Slot)
        Unplug(Slot, P)
    remove registered relationship Slot/PlugType
    send Deregistered event to slot host
```

- *Closed-Event*. The *CloseSlot* operation closes a slot and notifies the host.

```
CloseSlot(Slot)
    if(not Slot.IsOpen) return
    change state of Slot to closed
    send Closed event to slot host
    DeregisterAll(Slot)
```

- *Created-Event*. The *CreateUniqueExtension* and *CreateSharedExtension* operations create a contributor and notify the contributor before the contributor is plugged.

```
CreateUniqueExtension(ExtensionType)
    if(ExtensionType.IsSingleton and instance exists) return
    E = create extension meta object for ExtensionType
    E.Object = create .NET object for ExtensionType.Class
    send Created event to extension
    return E

CreateSharedExtension(ExtensionType)
    if(ExtensionType.IsSingleton and instance exists) return
    E = create extension meta object for ExtensionType
    make E shared instance of ExtensionType
    E.Object = create .NET object for ExtensionType.Class
```

send *Created* event to extension
for(all plugs *P* of extension *E*)
PT = type of *P*
for(all slots *S* where *PT* is registered)
if(*S.LazyLoad* and *S.AutoPlug* and not *S.Unique*)
Plug(S, P)
return E

- *Released-Event*. The *Release* operation closes the slots of the contributor and notifies the contributor before it is disposed.

Release(Extension)
UnplugWherePlugged(Extension)
for(all slots *S* of *Extension*)
CloseSlot(S)
send *Released* event to contributor
dispose .NET object
dispose instance meta object

3.9.2 The Core Extension

A plug-in-based application is broken up into a small core application which is extended by plug-in extensions. Extensions can be plugged into slots of the core application as well as into slots of other extensions. To bootstrap the process, the CM defines a singleton core extension with two slots for *Discovery* and *Startup*. Fig. 16 shows the core contract with the slot definitions for *Discovery* and *Startup*, and the core plug-in with the *Core* extension type.

A discoverer in Plux.NET is an extension and is not integral part of the composition infrastructure. To bootstrap, the composition infrastructure needs a discoverer to load the first application extensions. Among these first application extensions is often the desired real discoverer extension. A bootstrap discoverer is part of the core plug-in and it allows discovering a set of contracts and plug-ins. The names of contracts and plug-ins can be specified as arguments when the composition infrastructure is launched.

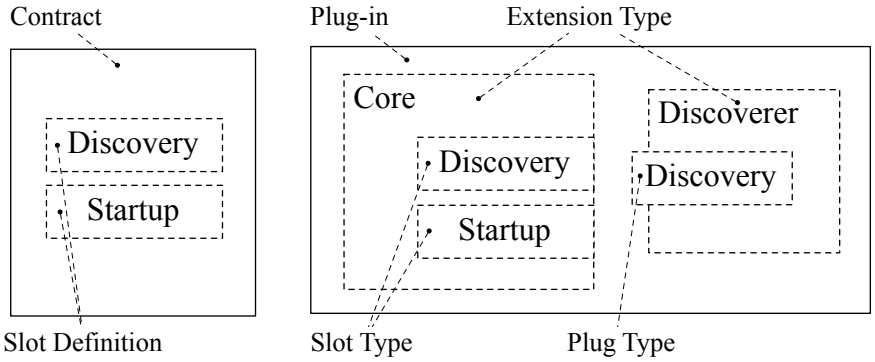


Figure 16. Contract and plug-in of core extension

3.9.3 Composing an Example Application

This section shows how the composition service composes an application. First, it creates the core application and the bootstrap discoverer. Then, the bootstrap discoverer discovers sample contracts and plug-ins, and the composition service composes the plug-ins. Finally, we show how to use a composition tool for manually replacing extensions and for shutting down the application by manually releasing the core extension.

Composing the Core Extension

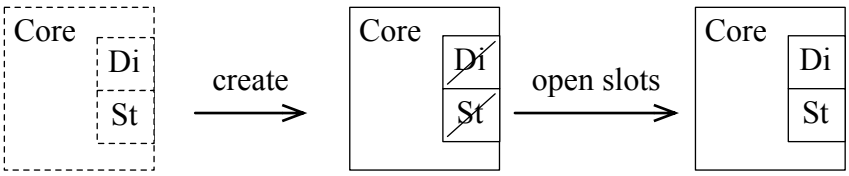
In the following pseudo code, the composition service starts the core application. It adds the core contract and the core plug-in to the CM and creates the core extension. The core extension cannot be plugged, because it is the root extension and does not have a plug. Instead, the startup and discovery slots are opened.

Add(Contract$_{\text{Core}}$*)*
Add(Plug-in$_{\text{Core}}$*)*

ExtensionType$_{\text{Core}}$ = extension type with name "Core"
Extension$_{\text{Core}}$ = *CreateSharedExtension(ExtensionType*$_{\text{Core}}$*)*
foreach(*Slot* in slots of *Extension*$_{\text{Core}}$)
    OpenSlot(Slot)

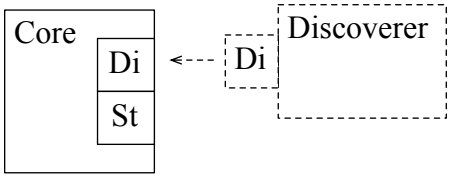
The log below shows the operations executed in the discovery and composition service when the pseudo code above is executed. The figure shows the corresponding composition result. The discovery service discovers the core contract and the core plug-in. Log messages of the discoverer service are marked with `[D]`. The composition service creates the core extension and opens its slots. Log messages of the composition service are marked with `[C]`.

```
[D] Contract discovered: "Core"
[D] Slot definition found: "Discovery"
[D] Slot definition found: "Startup"

[D] Plugin discovered: "Core"
[D] Extension type found: 1:"Core"
[D] Slot type found: "Discovery"
[D] Slot type found: "Startup"
[D] Extension type found 2:"Discoverer"
[D] Plug type found: "Discovery"

[C] Extension created: 1:"Core"
[C] Slot opened: 1:"Discovery" (Core)
[C] Slot opened: 2:"Startup" (Core)
```



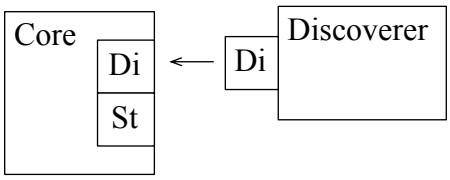
The composition service registers the `Discoverer` extension in the core.

```
[C] Extension type registered: 2:"Discoverer" >> 1:"Core" (Discovery)
```



The composition service creates the `Discoverer` extension and plugs it into the core. The bootstrap discoverer extension starts discovery now.

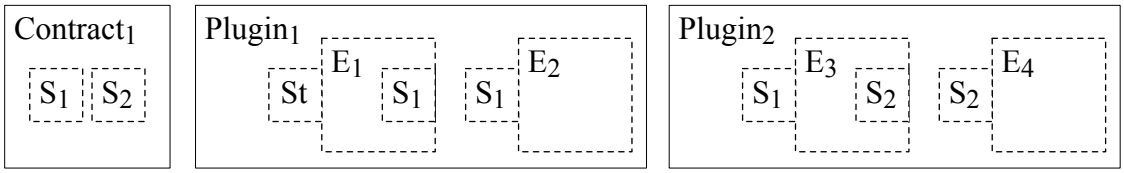
```
[C] Extension created: 2:"Discoverer"
[C] Extension plugged: 2:"Discoverer" >> 1:"Core" (Discovery)
```



Adding Extensions with the Bootstrap Discoverer

Let us assume that the discoverer now discovers a contract $Contract_1$ and two plug-ins $Plugin_1$ and $Plugin_2$. The discovery service finds the slot definitions from the contract and the extensions from the plug-ins.

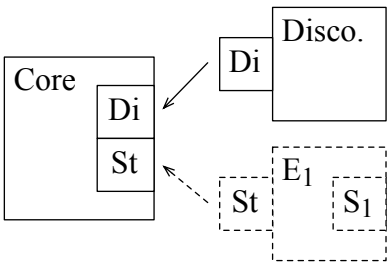
Add($Contract_1$)
Add($Plugin_1$)
Add($Plugin_2$)




```
[D] Contract discovered: "Contract1"
[D] Slot definition found: "S1"
[D] Slot definition found: "S2"
[D] Plugin discovered: "Plugin1"
[D] Extension type found: 3:"E1"
[D] Plug type found: "Startup"
[D] Slot type found: "S1"
[D] Extension type found: 4:"E2"
[D] Plug type found: "S1"
[D] Plugin discovered: "Plugin2"
[D] Extension type found: 5:"E3"
[D] Plug type found: "S1"
[D] Slot type found: "S2"
[D] Extension type found: 6:"E4"
[D] Plug type found: "S2"
```

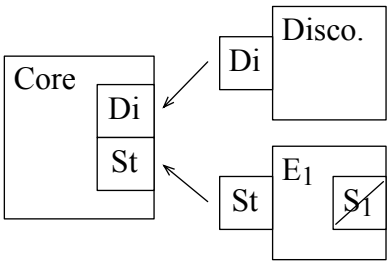
The composition service looks for extensions which contribute to open slots. It finds contributor E_1 and registers its startup plug in the core. The other extensions do not contribute to open slots.

```
[C] Extension type registered: 3:"E1" >> 1:"Core" (Startup)
```



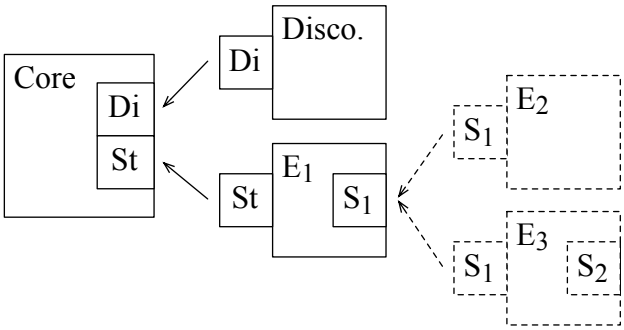
The composition service creates extension E_1 and plugs it into the core.

```
[C] Extension created: 3:"E1"
[C] Extension plugged: 3:"E1" >> 1:"Core" (Startup)
```



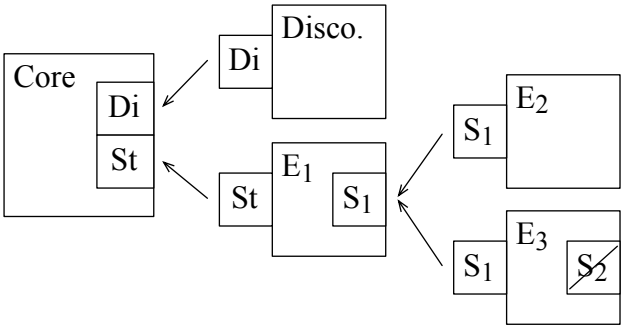
The composition service filled the slots of the core. Next it opens slot S_1 of host E_1 and registers contributors E_2 and E_3.

```
[C] Slot opened: 3:"S1" (E1)
[C] Extension type registered: 4:"E2" >> 3:"E1" (S1)
[C] Extension type registered: 5:"E3" >> 3:"E1" (S1)
```



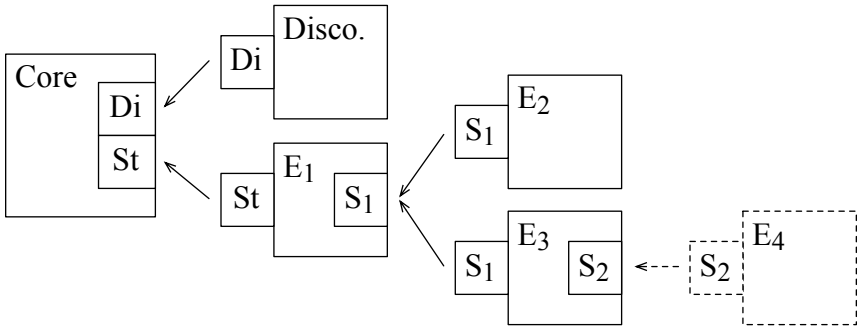
The composition services creates extension E_2 and E_3 and plugs them into host E_1.

```
[C] Extension created: 4:"E2"
[C] Extension plugged: 4:"E2" >> 3:"E1" (S1)
[C] Extension created: 5:"E3"
[C] Extension plugged: 5:"E3" >> 3:"E1" (S1)
```



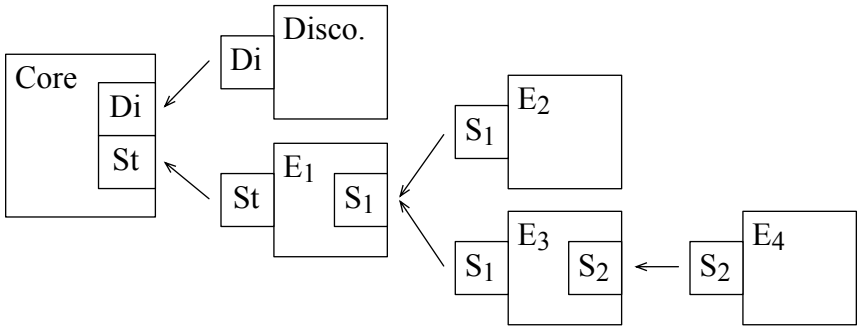
The composition service filled the slots of host E_1. Host E_2 does not have slots to open. Next the composition service opens slot S_2 of host E_3 and registers contributor E_4.

```
[C] Slot opened: 4:"S2" (E3)
[C] Extension type registered: 6:"E4" >> 5:"E3" (S2)
```



The composition services creates an extension E_4 and plugs it into host E_3.

```
[C] Extension created: 6:"E4"
[C] Extension plugged: 6:"E4" >> 5:"E3" (S2)
```

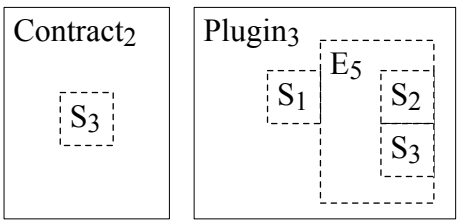


The composition service filled the slots of host E_3. Host E_4 does not have slots to open. All discovered extensions are composed.

Manually Replacing Extensions

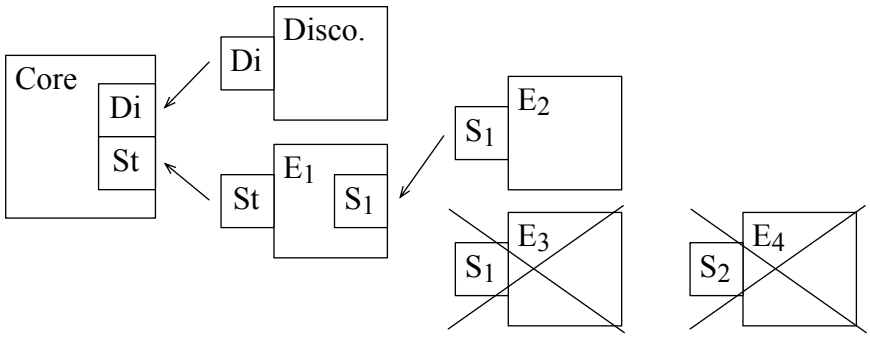
Let us assume that a composition tool deregisters extension E_3 and discovers contract $Contract_2$ and plug-in $Plugin_3$. Extension E_5 should replace extension E_3. Extension E_5 has a slot S_2, like extension E_3 did, and another slot S_3. The composition tool calls the following methods from the composition service API.

Deregister(ExtensionType$_3$)
Add(Contract$_2$)
Add(Plugin$_3$)



The composition service deregisters extension E_3. Before it releases extension E_3, it closes slot S_2, and deregisters extension E_4. After slot S_2 is closed, extension E_3 can be released and deregistered.

```
[C] Extension type deregistering: 5:"E3" << 3:"E1" (S1)
[C]   Extension unplugged: 5:"E3" << 3:"E1" (S1)
[C]   Extension releasing: 5:"E3"
[C]   Slot closing: 4:"S2" (E3)
[C]     Extension type deregistering: 6:"E4" << 5:"E3" (S2)
[C]       Extension unplugged: 6:"E4" << 5:"E3" (S2)
[C]       Extension releasing: 6:"E4"
[C]       Extension released: 6:"E4"
[C]     Extension type deregistered: 6:"E4" << 5:"E3" (S2)
[C]   Slot closed: 4:"S2" (E3)
[C]   Extension released: 5:"E3"
[C] Extension type deregistered: 5:"E3" << 3:"E1" (S1)
```



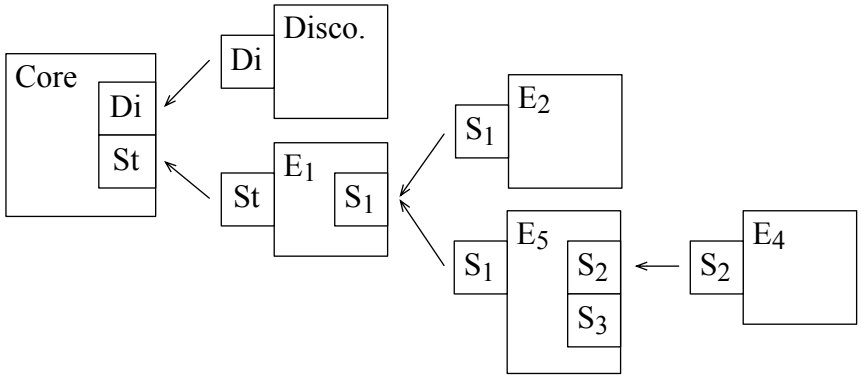
The composition tool adds $Contract_2$ and $Plugin_3$ to the CM. The discovery service finds slot definition S_3 and extension E_5.

```
[D] Contract discovered: "Contract2"
[D] Slot definition found: "S3"
```

```
[D] Plugin discovered: "Plugin3"
[D] Extension type found: "E5"
[D] Plug type found: "S1"
[D] Slot type found: "S2"
[D] Slot type found: "S3"
```

The composition service registers, creates and plugs contributor E_5 in host E_1. Next, it opens slots S_2 and S_3 of host E_5 and registers contributor E_4. Then the composition service creates an extension E_4 and plugs it into host E_5.

```
Extension type registered: 7:"E5" >> 3:"E1" (S1)
Extension created: 7:"E5"
Extension plugged: 7:"E5" >> 3:"E1" (S1)
Slot opened: 5:"S2" (E5)
Slot opened: 6:"S3" (E5)
Extension type registered: 6:"E4" >> 7:"E5" (S2)
Extension created: 8:"E4"
Extension plugged: 8:"E4" >> 7:"E5" (S2)
```



The composition service finished the replacement. It replaced extension E_3 with extension E_5. Since extension E_5, like extension E_3, opened a slot S_2, slot S_2 has been composed in the same way as before. However, the original contributor E_4 with id 6 has been released, and a new contributor E_4 with id 8 was created. If, for instance, the composition tool had disabled auto releasing for contributor E_4, the original contributor with id 6 would have been reused.

Manually Removing Extensions

Let us assume that a composition tool releases the core to shut down the application.

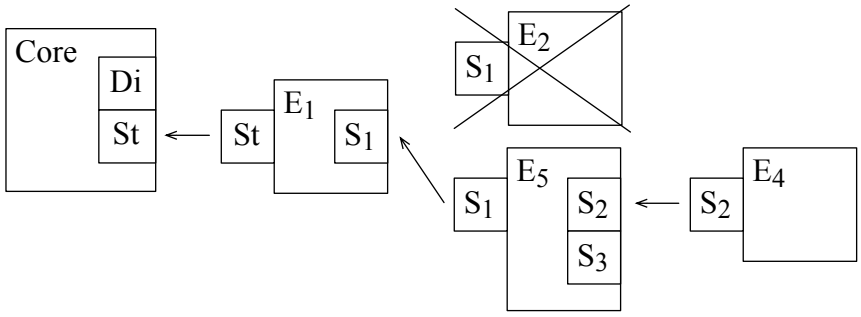
Release(Core)

The composition service closes the slots of the core extension. It closes the discovery slot and deregisters the discoverer. It closes the startup slot and deregisters contributor E_1.

```
[C] Extension releasing: 1:"Core"
[C] Slot closing: 1:"Discovery" (Runtime)
[C]   Extension type deregistering: 2:"Discoverer" << 1:"Core" (Discovery)
[C]     Extension unplugged: 2:"Discoverer" << 1:"Core" (Discovery)
[C]     Extension releasing: 2:"Discoverer"
[C]     Extension released: 2:"Discoverer"
[C]   Extension type deregistered: 2:"Discoverer" << 1:"Core" (Discovery)
[C] Slot closed: 1:"Discovery" (Runtime)
[C] Slot closing: 2:"Startup" (Runtime)
[C]     Extension type deregistering: 3:"E1" << 1:"Runtime" (Startup)
[C]     Extension unplugged: 3:"E1" << 1:"Runtime" (Startup)
[C]     Extension releasing: 3:"E1"
```

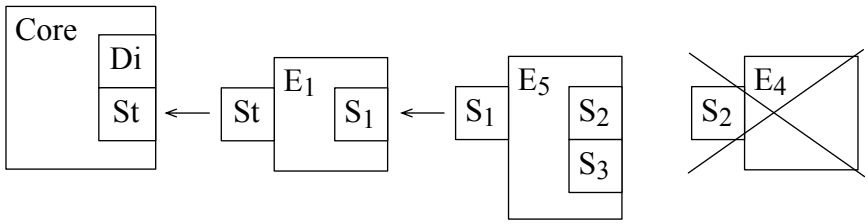
When host E_1 is releasing, the composition service closes its slot S_1 and deregisters contributor E_2 from Slot S_1. Host E_2 does not have slots.

```
[C]     Slot closing: 3:"S1" (E1)
[C]       Extension type deregistering: 4:"E2" << 3:"E1" (S1)
[C]         Extension unplugged: 4:"E2" << 3:"E1" (S1)
[C]         Extension releasing: 4:"E2"
[C]         Extension released: 4:"E2"
[C]       Extension type deregistered: 4:"E2" << 3:"E1" (S1)
```



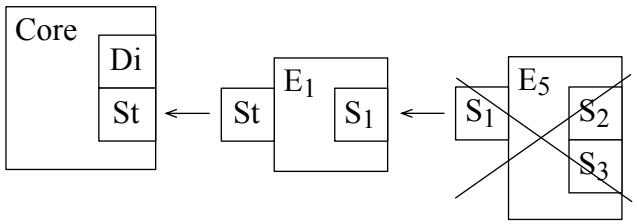
The composition service deregisters contributor E_5 and closes slot S_2 of host E_5. Next it deregisters contributor E_4.

```
[C]       Extension type deregistering: 7:"E5" << 3:"E1" (S1)
[C]         Extension unplugged: 7:"E5" << 3:"E1" (S1)
[C]         Extension releasing: 7:"E5"
[C]         Slot closing: 5:"S2" (E5)
[C]           Extension type deregistering: 6:"E4" << 7:"E5" (S2)
[C]             Extension unplugged: 8:"E4" << 5:"E3" (S2)
[C]             Extension releasing: 8:"E4"
[C]             Extension released: 8:"E4"
[C]           Extension type deregistered: 6:"E4" << 7:"E5" (S2)
[C]         Slot closed: 5:"S2" (E5)
```



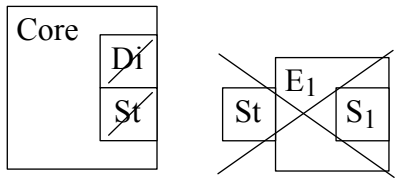
The composition service closes slot S_3 of host E_5. After slots S_2 and S_3 are closed, extension E_5 can be released and deregistered.

```
[C]         Slot closing: 6:"S3" (E5)
[C]         Slot closed: 6:"S3" (E5)
[C]         Extension released: 7:"E5"
[C]       Extension type deregistered: 7:"E5" << 3:"E1" (S1)
```



After slot S_1 is closed, contributor E_1 can be released and deregistered. After the startup slot is closed, the core can be released and the application exits.

```
[C]     Slot closed: 3:"S1" (E1)
[C]     Extension released: 3:"E1"
[C]   Extension type deregistered: 3:"E1" << 1:"Core" (Startup)
[C] Slot closed: 2:"Startup" (Core)
```




```
[C] Extension released: 1:"Core"
```

3.9.4 Queueing Composition Operations

The composition example in the previous section shows how the composition service composes an application in layers. It fills the slots of a host, before it opens the slots of the plugged contributors. Because the composition operations recursively call other composition operations, the extension graph would actually grow depth-first instead of breadth-first. For example, the *Plug* operation (see page 56) opens the slots of the plugged contributor. The *OpenSlot* operation registers all contributors, and the *RegisterAll* operation calls the *Plug* operation recursively.

Plug — OpenSlot — RegisterAll — Register — Plug

If composition would grow the application depth-first, a host extension could not react to a notification of a plugged contributor, before the contributor's slots are filled. The desired behavior is to grow the application breadth-first, i.e. to fill the slots of a host before the slots of the contributors are opened. To achieve that, the composition services serializes operations in a first-in-first-out (FIFO) queue. The affected operations are *OpenSlot*, *CloseSlot*, all *Register*, all *Plug*, and all *CreateAndPlug* operations.

Fig. 17 shows the FIFO queue resulting from the composition example from the previous section. In step 0, the composition service enqueued the operations *OpenSlot(*$Slot_{\text{Discovery}}$*)* and *OpenSlot(*$Slot_{\text{Startup}}$*),* when it started the composition.

In step 1, the composition service dequeues the *OpenSlot* operation for the discovery slot which enqueues the *RegisterAll(*$Slot_{\text{Discovery}}$*)* operation. Arrows indicate that an operation enqueues another operation. Then it dequeues the *OpenSlot* operation for the startup slot which enqueues the *RegisterAll(*$Slot_{\text{Startup}}$*)* operation.

When step 1 continues, the *RegisterAll(Slot)* operations enqueue *Register(Slot, PlugType)* operations, which in turn enqueue *Plug(Slot, PlugType)* operations. The plug operation propagates composition to contributor E_1, when the *OpenSlot(*S_1*)* operation is enqueued. The composition service processes steps 2 and 3 in the same way.

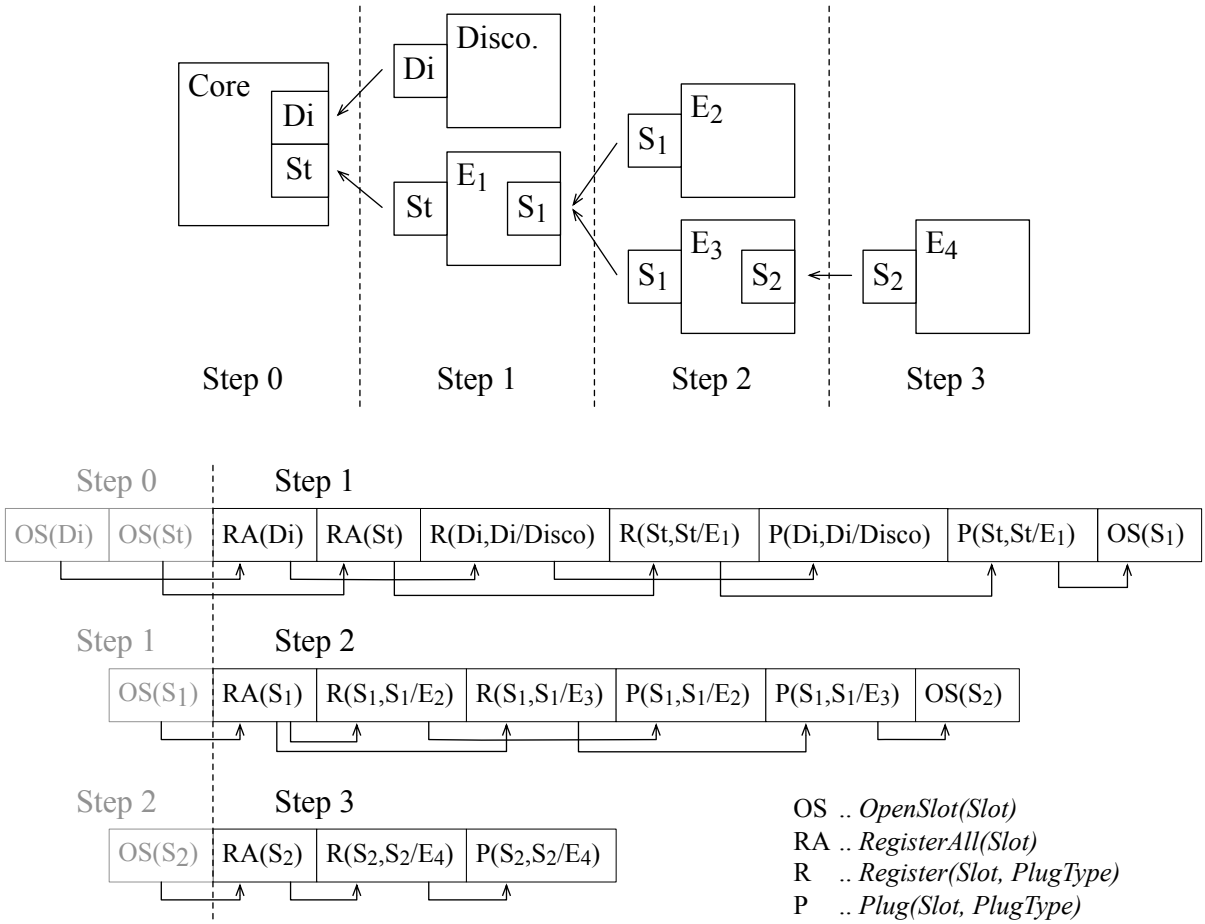


Figure 17. Task queue of the composition service

Chapter 4: Plux.NET Composition Infrastructure

This chapter describes a composition infrastructure (CI) which implements the composition model described in Chapter 3. The CI allows building rich client applications which support fine-grained customization and dynamic reconfiguration using plug-and-play composition. This chapter describes how the CI is designed.

This chapter is structured as follows: Section 4.1 describes the custom .NET attributes used for type meta element specification. Section 4.2 overviews the architecture of the Plux.NET CI. Sections 4.3 to 4.8 describe the components of the CI, i.e., the type store, the discovery core, the bootstrap discoverer, the assembly analyzer, the instance store, and the composition core.

4.1 Attributes for Type Meta Elements

In the Plux.NET CI, the default mechanism to specify type meta data in contract and plug-in assemblies are custom .NET attributes. Attributes are pieces of meta information which can be attached to language constructs such as classes, interfaces, methods or fields in the source code. At run time, the attributes attached to a language construct can be retrieved using reflection. In addition to pre-defined attributes in the .NET Framework, programmers can declare custom attributes. Plux.NET declares such attributes for type meta elements in the CI.

4.1.1 Attributes for Slot Definitions

The `SlotDefinition` attribute specifies a slot definition and can be attached to interfaces. It provides a member for a name. If multiple slot definition attributes are attached to one interface, all slot definitions share the same interface and parameter definitions. The `ParamDefinition` attribute specifies a parameter definition and can be attached to interfaces. It provides properties for a name, a .NET type, and a default value.

The example below defines a slot for menu commands. The interface `ICommand` has a `Do` method which is called when the user selects the command from the menu. With the `Text` parameter, every contributor provides a menu command string, which is used by the host to build and display the menu.

```
[SlotDefinition("Command")]
[ParamDefinition("Text", typeof(String), "(unavailable)"]
interface ICommand {
  void Do();
}
```

Table 6 summarizes the Plux.NET attributes used for slot definitions.

[SlotDefinition(name)] attachable to interfaces

Syntax

```
"[" "SlotDefinition" "(" name ")" "]"
```

Arguments

name	a string specifying a user-defined name for the slot definition

[ParamDefinition(name, properties)] attachable to interfaces

Syntax

```
"[" "ParamDefinition" "(" name "," type [ "," default ] ")" "]"
```

Arguments

name	a string specifying a user-defined name for the parameter definition
type	a .NET type specifying the expected data type for the parameter
default	a constant which is used as a default value if the contributor omits the parameter

Table 6. Plux.NET attributes for slot definitions

4.1.2 Attributes for Contributor Extensions

The `Extension` attribute specifies an extension and can be attached to classes. It provides properties for a name and for the singleton and auto releasing settings. The properties `OnCreated` and `OnReleased` specify event handlers for composer events (see page 55).

The `Plug` attribute specifies a plug and can be attached to extension classes. It provides properties for a name and for the auto registering and auto plugging settings. The properties `OnPlugged` and `OnUnplugged` specify event handlers for composer events.

The `Param` attribute specifies a parameter and can be attached to extension classes. It provides properties for a name, for a value object, and optionally the name of a plug which this parameter value is associated to. Multiple parameter attributes can be attached to a class.

The example below defines a contributor for the `Command` slot. The contributor class `CloseCommand` will appear in the menu with the command string `"Close"`. The contributor is not a singleton and will be automatically registered, plugged, and released. The contributor registers event handler methods for when it is created or released, and for when it is plugged or unplugged.

```
[Extension("CloseCommand", Singleton=false, AutoRelease=true,
  OnCreated="HandleCreated", OnReleased="HandleReleased")]
[Plug("Command", AutoRegister=true, AutoPlug=true,
  OnPlugged="HandlePlug", OnUnplugged="HandleUnplug")]
[Param("Text", "Close", Plug="Command")]
public class CloseCommand : ICommand { ... }
```

Table 7 summarizes the Plux.NET attributes used for contributor extensions:

[Extension(name, properties)] attachable to classes

Syntax
```
"[" "Extension" [ "(" ( name | property ) { "," property } ")" ] "]"
```
Arguments

name	a string specifying a user-defined name for the extension; if omitted the class name is used as extension name
Singleton=true	if true, only one instance of this extension can be created (default: false)
AutoRelease=true	if true, the extension is automatically released after it was unplugged (default: true)
OnCreated=meth	a method name specifying a handler for the *Created* event
OnReleased=meth	a method name specifying a handler for the *Released* event

[Plug(name, properties)] attachable to classes

Syntax
```
"[" "Plug" "(" name { "," property } ")" "]"
```
Arguments

name	a string specifying the name of the slot into which this plug fits
AutoPlug=true	if true, the extension is automatically plugged after it was registered (default: true)
AutoRelease=true	if true, the extension is automatically released after it was unplugged (default: true)
OnPlugged=meth	a method name specifying a handler for the *Plugged* event
OnUnplugged=meth	a method name specifying a handler for the *Unplugged* event

[Param(name, properties)] attachable to classes

Syntax
```
"[" "Param" "(" name "," value [ "," plug ] ")" "]"
```
Arguments

name	a string specifying the name of the parameter
value	an object specifying a parameter value
plug	a string specifying the name of the plug to which this parameter belongs to; if omitted the parameter applies to all plugs of the extension

Table 7. Plux.NET attributes for contributor extensions

Using the default values of the attribute properties the above example can be written as:

```
[Extension]
[Plug("Command", OnPlugged="HandlePlug", OnUnplugged="HandleUnplug")]
[Param("Text", "Close")]
public class CloseCommand : ICommand { ... }
```

4.1.3 Attributes for Host Extensions

The same `Extension` attribute is used for host extensions as well as for contributor extensions, because typically extensions have both roles, the role of a contributor and the role of a

host. The `Slot` attribute specifies a slot and can be attached to host extension classes. It provides properties for a name, for a slot definition, and for configuration settings. The properties `OnRegistered`, `OnPlugged`, etc. specify handlers for composer events.

The example below defines a host with a `Command` slot. The menu host opens a slot with the name `Command` which uses the slot definition `Command`. The slot accepts multiple shared contributors, and it loads contributors lazily. The menu host handles the *Registered* event, where it reads the `Text` parameter and builds the menu item. When the user selects the menu command, the menu host plugs the shared extension. Finally, the menu host handles the *Plugged* event, where it calls the `Do` method from interface `ICommand` (see Section 5.4.1 on page 114 for an example how to implement such a host).

```
[Extension("Menu")]
[Plug("Startup")]
[Slot("Command", Multiple=true, Unique=false,
  AutoRegister=true, AutoPlug=true, LazyLoad=true,
  OnRegistered="Command_Registered", OnPlugged="Command_Plugged", ...)]
public class MenuHost : IStartup { ... }
```

Table 8 summarizes the Plux.NET slot attribute for host extensions.

[Slot(name, properties)] attachable to classes

Syntax

```
"[" "Slot" "(" name { "," property } ")" "]"
```

Arguments

name	a string specifying the name of the slot
Definition="Name"	a string specifying the name of the slot definition; if omitted, the slot name specifies the slot definition
Multiple=true	if true, the slot allows registering multiple contributors (default: false)
Unique=true	if true, the slot requires unique contributors (default: false)
AutoOpen=true	if true, the slot is opened when the extension is plugged (default: true)
AutoRegister=true	if true, contributors are automatically registered after the slot was opened (default: true)
LazyLoad=true	if true, contributors are not automatically plugged after they were registered; when a host manually plugs a shared contributor, it is automatically plugged in other sharing hosts (default: false)
AutoPlug=true	if true, contributors are automatically plugged after they were registered (default: true)
AutoRelease=true	if true, contributors are automatically released after they were unplugged (default: true)
OnOpened=meth	a method name specifying a handler for the *Opened* event
OnRegistered=meth	a method name specifying a handler for the *Registered* event
OnPlugged=meth	a method name specifying a handler for the *Plugged* event
OnSelected=meth	a method name specifying a handler for the *Selected* event
OnDeselected=meth	a method name specifying a handler for the *Deselected* event
OnUnplugged=meth	a method name specifying a handler for the *Unplugged* event
OnDeregistered=meth	a method name specifying a handler for the *Deregistered* event
OnClosed=meth	a method name specifying a handler for the *Closed* event

Table 8. Plux.NET attribute for host extensions

4.2 Architecture Overview

The Plux.NET composition infrastructure allows executing plug-ins which conform to the Plux.NET composition model. Fig. 18 shows the components of the CI and their interactions. This section contains a short description for each component. Subsequent sections contain detailed specifications.

To bootstrap composition, the CI loads an *bootstrap discoverer* when it starts. The bootstrap discoverer adds contracts and plug-ins which are specified as command line argument to the type store. It uses the *assembly analyzer* to extract type meta elements from custom .NET attributes in contract and plug-in assemblies. In typical scenarios, the bootstrap discoverer loads a custom discoverer, which subsequently discovers the remainder of the application's extensions.

The *type store* maintains types and acts as an observable object notifying about changes. Discoverer extensions, such as the bootstrap discoverer, use the *type builder* interface to create type meta elements for the type store. They use the *type store modifier* interface to add or remove types. Other components, such as the composition core, use the *type store reader* interface to read types. The composition core also uses the *type qualifier* interface to qualify types before it uses them in composition. The *type store observable* interface allows the composition core to register a change listener. Upon changes in the type store, the composition core automatically registers added types and deregisters removed types.

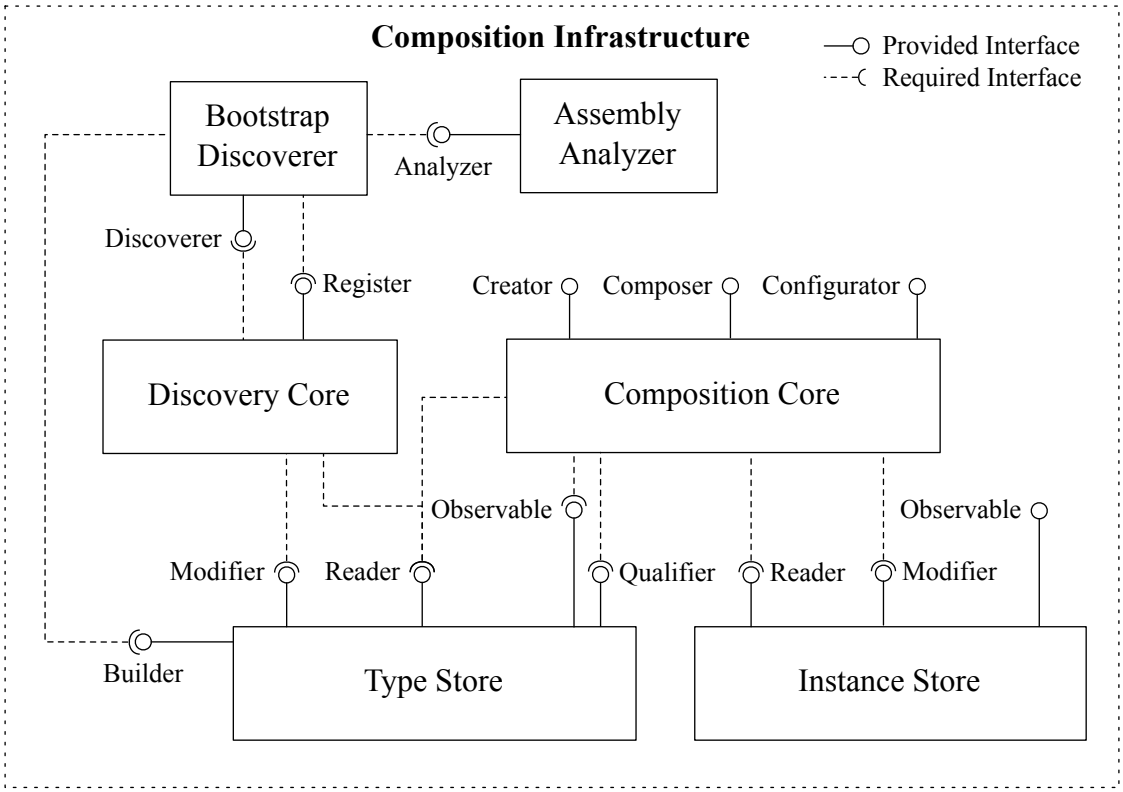


Figure 18. Architecture of the Plux.NET composition infrastructure

The *discovery core* provides the infrastructure that is necessary to discover types. The actual discovery mechanism is not part of the CI. Instead, the discovery core integrates external discoverer extensions and provides the *type builder* interface of the type store to them.

The *instance store* maintains instances and their relationships. It acts as an observable object notifying observers about changes. The composition core uses the *instance store modifier* interface to add or remove instances as well as relationships between instances. It also uses the *instance store reader* interface to determine requirements and provisions of hosts and contributors. The *instance store observable* interface is the place where host extensions register their change listeners when they react to the events of the composition core. The *instance store* notifies contributors when they are created and released, and it notifies hosts and contributors when composition relationships change, for example when contributors become known for a host.

The *composition core* creates instances and controls relationships between instances. It uses the *type store reader* to determine requirements and provisions of hosts and contributors. It uses the *type qualifier* to qualify types prior to their use in composition. The composition core provides the *creator* interface with operations for creating extensions and the *composer* interface with operations for controlling relationships. It also provides the *composition configurator* interface that allows configuring the composition procedure. The composition core uses the *instance store modifier* to store instances and their relationships. It also uses the *type store observable* to monitor the type store. When an extension is added/removed to/from the type store, the composition core registers/deregisters the extension in the instance store.

Composition tools use the *composition configurator* to configure the composition procedure through settings. They deactivate automatic composition operations and switch from the automatic mode to the manual mode. In manual mode, a composition tool can use the *creator* interface to create extensions, and the *composer* interface to change relationships.

4.3 Type Store

The *type store* maintains *type meta objects* (short: *types)*. The CI reads contracts and plug-ins from .NET assemblies. An assembly is the smallest unit for loading, deployment, and versioning in .NET (ECMA 2006). Contracts and plug-ins are library assemblies with the file extension *.dll. The type store maintains meta objects about types which are stored in .NET assemblies.

Fig. 19 shows a class diagram for contract and plug-in types. The class diagram extends the composition model class diagram from Chapter 3 (see page 35). Classes for contract types use the postfix *Info* or *Definition* in their name. Classes for extension types use the postfix *Info* in their name. The classes in the type store take the attributes from the composition model and explicitly add attributes for bi-directional navigation between meta elements. For example, the *SlotDefinitions* attribute of a contract definition contains a reference to all slot definitions, and each contained slot definition has a reference back in its *Contract* property. The class diagram also adds attributes for the event handlers, which extensions use to react to composer events. Each class implements the *MetaElement* interface and provides a name and an identification number. Identification numbers are consecutively numbered per kind.

The value of the *SlotDefinition* attribute in the plug type and slot type classes depends on the qualification state of the type. It can only be accessed if the type has been qualified before. Otherwise, the attribute raises an invalid operation exception. This is not relevant for application developers, because a host extension can only see registered contributors, and registered contributors are qualified by definition. However, the composition core, or composition tools must check the qualification state, before accessing the slot definition.

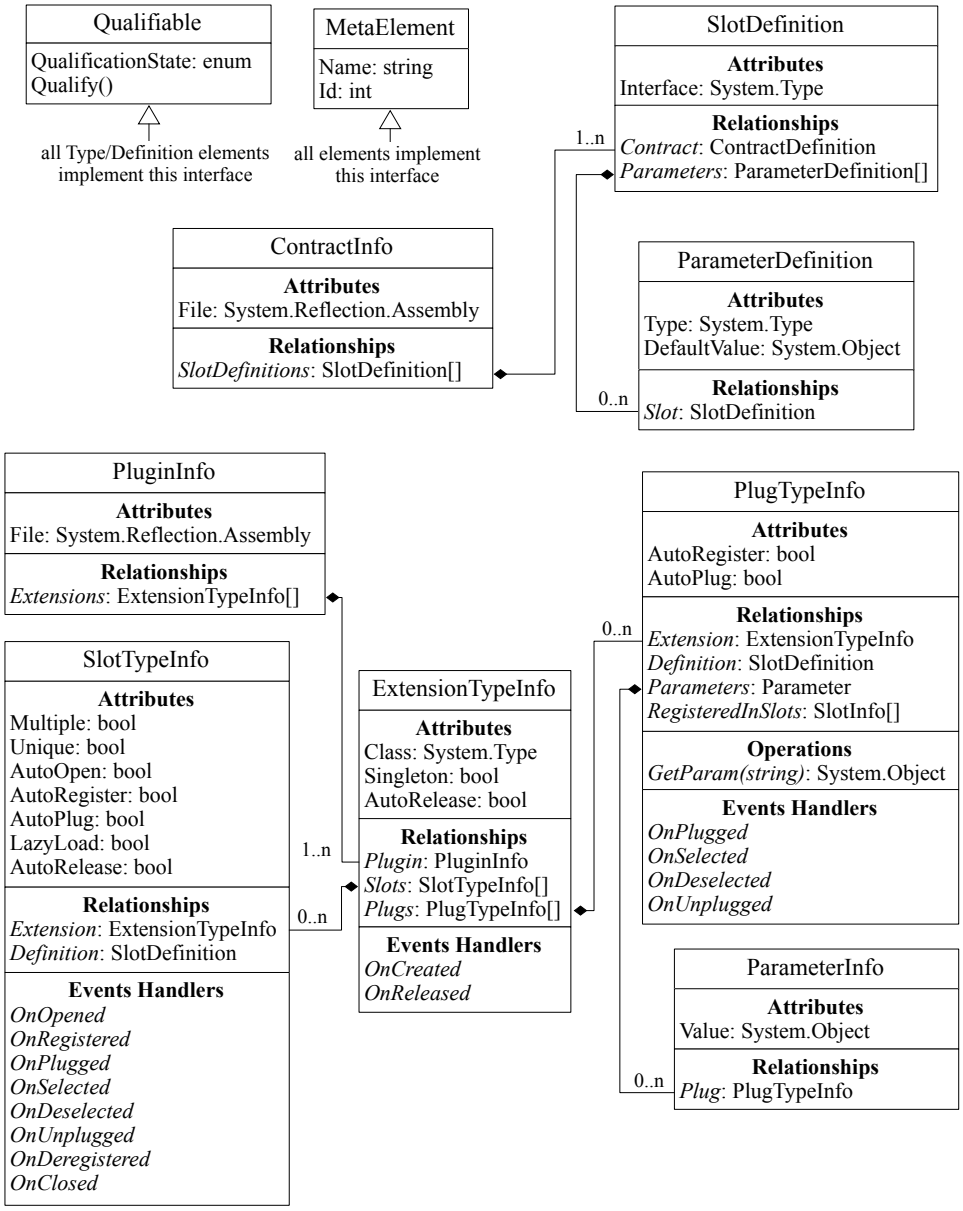


Figure 19. Class diagram of meta elements in the type store

The type store adds the following attributes and operations to support typical scenarios: Firstly, a host extension typically retrieves a parameter value by its name. Thus the plug type class provides the *GetParameter* operation which returns the parameter value for a given name. If a plug type does not specify a parameter, it returns the default value from the parameter definition. If the plug is not qualified, the method raises an unsupported operation exception. Sec-

ondly, the *RegisteredInSlots* enumeration contains all slots where a plug is registered. The registered slots are retrieved from the instance store (see page 82).

4.3.1 Type Qualifier Interface

The type qualifier interface allows the composition core to qualify types prior to their use in composition (see page 36). The composition core qualifies lazily, i.e. it does not qualify until a type is registered in a slot.

Every type in the type store implements the `Qualifiable` interface. The `Qualify` method tries to qualify a type. The property `IsQualificationMissing` is true when the type has not been tried. The property `IsQualified` is true if the type has successfully qualified. The property `Errors` returns error flags which indicate why a type did not qualify.

```
enum QualificationErrors {
  NameAlreadyExists=0x1, SlotDefinitionNotFound=0x2,
  InterfaceNotImplemented=0x4, ... }

interface Qualifiable {
  QualificationErrors Errors { get; }
  void Qualify();
  bool IsQualified { get; }
  bool IsQualificationMissing { get; }
}
```

For efficiency reasons, the composition core does not try to qualify failed types again. The following pattern makes sure that qualification is only tried once.

```
bool TryQualify(Qualifiable q) {
  if(q.IsQualificationMissing) {
    q.Qualify();
    if(!q.IsQualified) PrintNotQualifiedMessage(q.QualificationState);
  }
  return q.IsQualified;
}
```

When the content of the type store changes, the qualification state of types can change (see page 36). If a type references other types, and those other types are added/removed to/from the type store, the type store resets the qualification state to `QualificationMissing` and triggers a new qualification try.

4.3.2 Type Store Reader Interface

The type store reader interface allows the composition core and the discovery core to read types from the type store. The iterator `Contracts` contains all contracts; the iterator `Plugins` contains all plug-ins. The contracts and plug-ins provide iterator properties which allow navigating through their child elements (see Fig. 18 on page 73).

```
interface TypeStoreReader {
  IEnumerable<ContractInfo> Contracts { get; }
  IEnumerable<PluginInfo> Plugins { get; }
  // convenience properties
  IEnumerable<SlotDefinition> SlotDefinitions { get; }
  IEnumerable<ExtensionTypeInfo> ExtensionTypes { get; }
```

```
  IEnumerable<PlugTypeInfo> PlugTypes { get; }
  IEnumerable<SlotTypeInfo> SlotTypes { get; }
}
```

For convenient access, the reader provides properties which allow reading all slot definitions, extension types, slot types, and plug types in the type store with a global iterator. For example, the iterator *SlotDefinitions* contains all slot definitions of all contracts in the type store.

4.3.3 Type Store Observable Interface

The type store observable interface allows other components, such as the composition core, to register a change listener. The type store notifies registered observers after a contract or plug-in was added or removed.

```
interface TypeStoreObservable {
  event ContractEventHandler ContractAdded;
  event ContractEventHandler ContractRemoved;
  event PluginEventHandler PluginAdded;
  event PluginEventHandler PluginRemoved;
}
```

The observer registers event handler methods. When the type store fires an event, it includes which contract or plug-in has changed as an argument.

```
delegate void ContractEventHandler(object s, ContractEventArgs args);
class ContractEventArgs : System.EventArgs {
  ContractInfo Contract { get; internal set; }
}
delegate void PluginEventHandler(object s, PluginEventArgs args);
class PluginEventArgs : System.EventArgs {
  PluginInfo Plugin { get; internal set; }
}
```

4.3.4 Type Builder Interface

The type builder interface allows discoverer extensions to create type meta elements. The discoverer extracts the meta data from the contract and plug-in assemblies, builds the corresponding type meta elements using the type builder, and passes the types to the discovery core.

```
interface TypeBuilder {
  ContractInfo CreateContract(SlotDefinition[] slots);
  SlotDefinition CreateSlotDefinition(string name, System.Type interface,
      ParameterDefinition[] params);
  ParameterDefinition CreateParameterDefinition(string name,
      System.Type type, object defaultValue);
  PluginInfo CreatePlugin(
      string name, IEnumerable<ExtensionTypeInfo> extTypes);
  ExtensionTypeInfo CreateExtensionType(
      string name, System.Type clazz, SlotTypeInfo[] slots,
      PlugTypeInfo[] plugs, bool IsSingleton);
  SlotTypeInfo CreateSlotType(string name, SlotDefinition slot);
  Parameter CreateParameter(string name, object value);
  PlugTypeInfo CreatePlugType(string name, SlotDefinition slot,
```

```
        Parameter[] params);
}
```

4.3.5 Type Store Modifier Interface

The type store modifier interface allows the discovery core to add and remove types in the type store. The `Add` operation adds contracts or plug-ins to the type store; the `Remove` operation removes contracts or plug-ins from the type store. The `GetTypeBuilder` method allows the discoverer to access the type builder.

```
interface TypeStoreModifier {
  void Add(ContractInfo[] contracts, PluginInfo[] plugins);
  void Remove(ContractInfo[] contracts, PluginInfo[] plugins);
  TypeBuilder GetTypeBuilder();
}
```

After the type store processed an `Add` or `Remove` call, it notifies registered observers about the changes. Upon that notification, the composition core will update the composition with the changed types. If a discoverer wants to make sure that multiple contracts and plug-ins are added to the type store before the composition is updated, it must add those contracts and plug-ins within a single call of the `Add` method. This is important for qualification, because then all slot definitions will be available in the type store, when the extensions are tried to be qualified. To allow a discoverer to add multiple contracts and plug-ins in a single call, the add and remove methods accept arrays as arguments. This reasoning does not apply when types are removed, because the sequence of removal is irrelevant. The `Remove` method implements the same pattern anyway, for reasons of symmetry and convenience.

The following code shows how this pattern is implemented in the type store. At first, the `Add` method adds all contracts to the type store, then it adds the plug-ins, and finally it notifies the observers.

```
class TypeStore : TypeStoreModifier, ... {
  public void Add(ContractInfo[] contracts, PluginInfo[] plugins) {
    foreach(ContractInfo c in contracts) StoreType(c);
    foreach(PluginInfo p in plugins) StoreType(p);
    foreach(ContractInfo c in contracts)
      OnContractAdded(this, new ContractEventArgs { Contract = c; });
    foreach(PluginInfo p in plugins)
      OnPluginAdded(this, new PluginEventArgs { Plugin = p; });
  }
  private void StoreType(MetaElement elem) { /*not shown*/ }
  protected virtual void OnContractAdded(object s, ContractEventArgs args) {
    if(ContractAdded != null)
      ContractAdded(s, args);
  }
  ...
}
```

Following a convention in the .NET Framework, the class provides an `OnContractAdded` method to raise the `ContractAdded` event. Table 9 shows all notification methods in the type store.

Type Modifier Method	Type Observer Event	Event Args
Add	ContractAdded	ContractEventArgs
Add	PluginAdded	PluginEventArgs
Remove	ContractRemoved	ContractEventArgs
Remove	PluginRemoved	PluginEventArgs

Table 9. Type store notifications

4.4 Discovery Core

Discovery is customizable in the Plux.NET composition infrastructure. The *discovery core* integrates discoverer extensions into the CI. Custom discoverers typically comprise two parts. The *discoverer* part detects when contracts of plug-ins are added to or removed from a monitored component repository. The *analyzer* part extracts type meta data from the assemblies. The discoverer passes the type meta data to the discovery core. The discovery core stores the type meta data into the type store.

The CI provides a bootstrap discoverer (see page 80) and an assembly analyzer for .NET attributes (see page 81). The customizable discovery architecture allows replacing the discoverer, e.g. by discovery from multiple directories, discovery over the web, or discovery by user dialog. The assembly analyzer can also be replaced, e.g. to analyze XML files, or to analyze a custom composition language.

4.4.1 Discoverer Interface

A discoverer extension must implement the `Discoverer` interface. The discovery core calls `StartDiscovery` to start the discoverer and passes the type builder. The discovery core registers an event handler in the `DiscovererEvent` of the discoverer. Upon changes in the component repository, the discoverer calls back the discovery core.

```
[SlotDefinition("Discovery")]
interface Discoverer {
  void StartDiscovery(TypeBuilder builder);
  event DiscoveryEventHandler DiscoveryEvent;
}
```

When discovery starts, the discoverer should perform an initial discovery run. It should discover all types which are currently discoverable in the component repository. After initial discovery, the discoverer should continue to monitor the component repository and notify the discovery core upon changes.

Upon initial discovery and also upon later changes, the discoverer should always collect all changed types to a batch. It should build types for all changed assemblies and fire one single discovery event. Together with the event the discoverer sends `DiscoveryEventArgs` containing the type of change and the changed types.

```
delegate DiscoveryEventHandler(object s, DiscoveryEventArgs args);

enum DiscoveryChangeType { Add, Remove }

class DiscoveryEventArgs : System.EventArgs {
  DiscoveryChangeType ChangeType { get; set; }
  ContractInfo[] Contracts { get; set; }
  PluginInfo[] Plugins { get; set; }
}
```

4.4.2 Discovery Registrar Interface

The discovery registrar interface allows discoverer extensions to register or deregister in the discovery core. In this context, registration means to store a reference to the discoverer in the discovery core. This should not be confused here with the registered relationship in the composition model.

```
interface DiscoveryRegistrar {
  void Register(Discoverer discoverer);
  void Unregister(Discoverer discoverer);
}
```

When a discoverer registers, the discovery core connects its event handler method to the discoverer, then calls `StartDiscovery`, and passes the type builder. When a discoverer unregisters, the discovery core disconnects his event handler. When the discovery extension fires a discovery event, the discovery core updates the type store.

```
class DiscoveryCore : DiscoveryRegistrar {
  TypeStoreModifier typeStore = GetTypeStore();
  public void Register(Discoverer discoverer) {
    discoverer.DiscoveryEvent += OnDiscoveryEvent;
    discoverer.StartDiscovery(typeStore.GetTypeBuilder());
  }
  public void Unregister(Discoverer discoverer) {
    discoverer.DisocveryEvent -= OnDiscoveryEvent;
  }
  void OnDiscoveryEvent(object s, DiscoveryEventArgs args) {
    switch(args.ChangeType) {
      case Add: typeStore.Add(args.Contracts, args.Plugins); break;
      case Remove: typeStore.Remove(args.Contracts, args.Plugins); break;
    }
  }
  ...
}
```

4.5 Bootstrap Discoverer

The bootstrap discoverer extension discovers a set of contracts and plug-ins, the names of which were specified as command line arguments when the CI was launched. The bootstrap discoverer class implements the `Discoverer` interface (see page 79). It expects an array of assembly names and a type analyzer.

```
[Extension]
[Plug("Discovery")]
class BootstrapDiscoverer : Discoverer {
  public BootstrapDiscoverer(string[] assemblies, Analyzer analyzer) { ... }
  public void StartDiscovery(TypeBuilder builder);
  public event DiscoveryEventHandler DiscoveryEvent;
}
```

A type analyzer provides meta data for .NET assemblies. The `GetContracts` method reads contracts from an assembly, the `GetPlugins` method reads plug-ins from an assembly. The `AnalyzeFile` method combines both operations in one method and reads contracts as well as plug-ins.

```
[SlotDefinition("Analyzer")]
interface Analyzer {
  ContractInfo[] GetContracts(string file, TypeBuilder builder);
  PluginInfo[] GetPlugins(string file, TypeBuilder builder);
  bool AnalyzeFile(string file, TypeBuilder builder,
      out ContractInfo[] contracts, out PluginInfo[] plugins);
}
```

When the bootstrap discoverer is started by the discovery core, it processes the array with the assembly names. It builds types using the type analyzer and buffers them in collections. After the discoverer finished analyzing the assemblies, it fires the discovery event and passes the *Add* change type and the collected types as event arguments.

```
  void StartDiscovery(TypeBuilder builder) {
    var allContracts = new List<ContractInfo>();
    var allPlugins = new List<PluginInfo>();
    foreach(string file in assemblies) {
      allContracts.AddRange(analyzer.GetContracts(file, builder));
      allPlugins.AddRange(analyzer.GetPlugins(file, builder));
    }
    OnDiscoveryEvent(this, new DiscoveryEventArgs() {
        ChangeType = DiscoveryChangeType.Add;
        Contracts = allContracts.ToArray(typeOf(ContractInfo));
        Plugins = allPlugins.ToArray(typeOf(PluginInfo));
    });
  }
}
```

4.6 Assembly Analyzer

The assembly analyzer extension analyzes contract and plug-in assemblies and extracts type meta data from Plux.NET attributes. It provides a plug for the *Analyzer* slot and can be reused by custom discoverer extensions.

```
[Extension]
[Plug("Analyzer")]
public class AssemblyAnalyzer : Analyzer { ... }
```

At startup, the CI creates the bootstrap discoverer, passes the assembly analyzer, and registers the bootstrap discoverer in the discovery core.

```
void Bootstrap(string[] args) {
  DiscoveryRegistrar registrar = GetDiscoveryRegistrar(); /*not shown*/
  registrar.Register(new BootstrapDiscoverer(args, new AssemblyAnalyzer());
}
```

Fig. 20 shows the discoverer configuration used for bootstrapping.

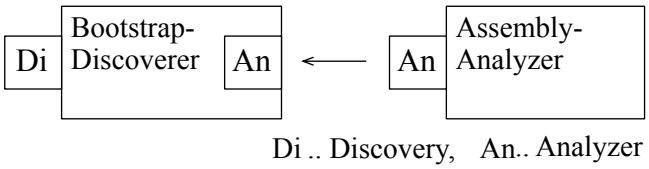


Figure 20. Bootstrap discoverer and assembly analyzer

4.7 Instance Store

The *instance store* maintains *instance meta elements* (short: *instances*) and their relationships. Relationships can be grouped in two categories. Firstly, the *Registered* relationship connects a slot in the instance store with a plug type in the type store. Because the *registered* relationship references the type store, the instance store is a type store observer and is notified when a type is removed. Secondly, the *Plugged* and *Selected* relationships connect slots and plugs within the instance store.

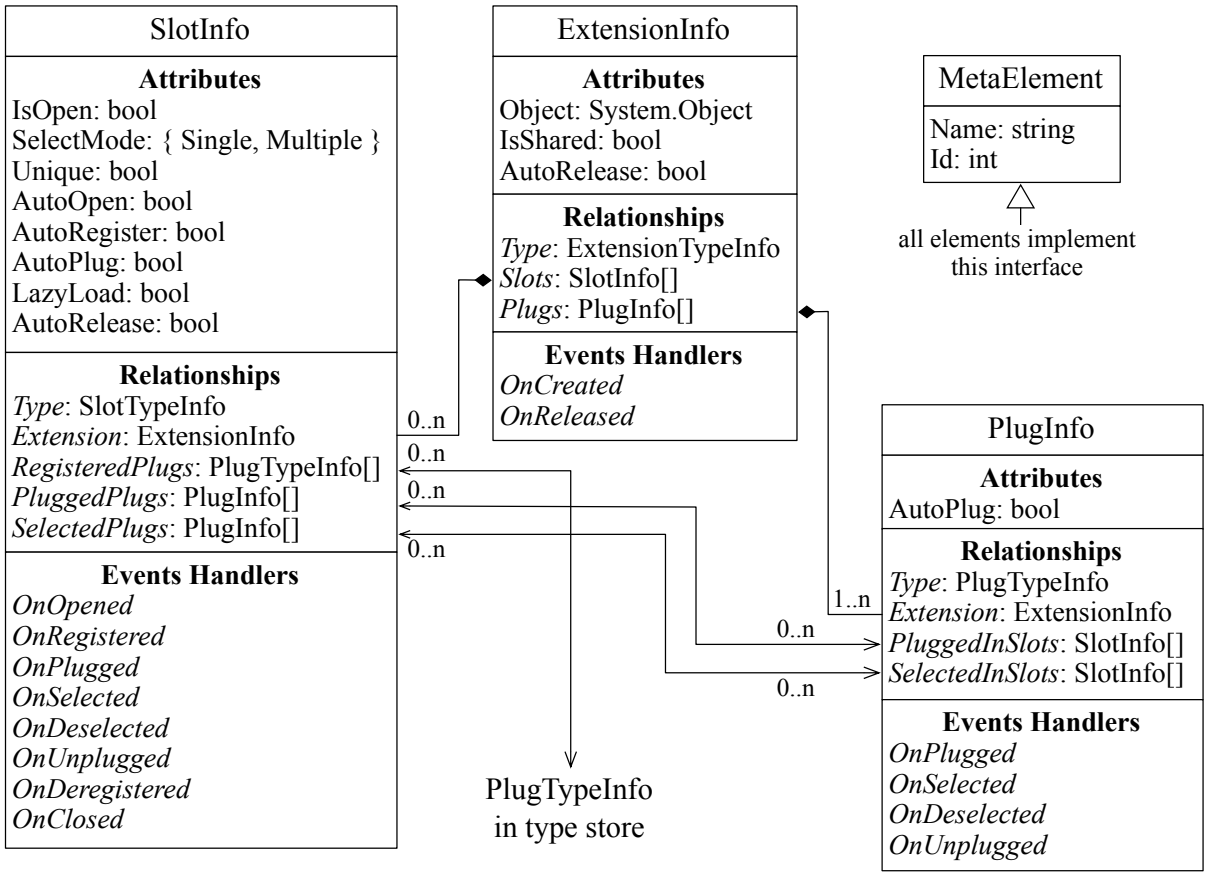


Figure 21. Class diagram of meta elements in the instance store

Fig. 21 shows a class diagram for instances. The class diagram extends the composition model class diagram from Chapter 3 (see page 35). In contrast to the class names from the composition model, the class names classes in the instance store use the postfix *Info*, adhering to a convention in the .NET Framework for meta classes. The classes in the instance store take the attributes from the composition model and explicitly add attributes for bi-directional navigation between meta elements. For example, the *Slots* attribute of an extension contains a reference to all slots, and each slot has a reference back in its *Extension* attribute.

The classes allow navigation along composition relationships. The *RegisteredPlugs* attribute in class *SlotInfo*, and the *RegisteredInSlots* attributes in class *PlugTypeInfo* (see Fig. 19 on page 75) contain the *registered* relationships. The *PluggedPlugs* attribute in class *SlotInfo*, and the *PluggedInSlots* attribute in class *PlugInfo* contain the *plugged* relationship. The *SelectedPlugs* attribute in class *SlotInfo*, and the *SelectedInSlots* attribute in class *PlugInfo* contain the *selected* relationship.

The class diagram also adds attributes for the event handlers, which extensions use to react to composer events. Each class implements the *MetaElement* interface and provides a name and an identification number. Identification numbers are consecutively numbered per meta element type.

4.7.1 Instance Store Reader Interface

The instance store reader interface allows the composition core to read instances from the instance store. The iterator `Extensions` contains all extensions.

```
interface InstanceStoreReader {
  IEnumerable<ExtensionInfo> Extensions { get; }
  IEnumerable<SlotInfo> Slots { get; }
  IEnumerable<PlugInfo> Plugs { get; }
  ExtensionInfo GetExtension(object dotNetObj);
  SlotInfo GetSlot(object dotNetObj, string name);
  PlugInfo GetPlug(object dotNetObj, string name);
  bool IsRegistered(SlotInfo slot, PlugTypeInfo plugType);
  bool IsPlugged(SlotInfo slot, PlugInfo plug);
  bool IsSelected(SlotInfo slot, PlugInfo plug);
}
```

For convenient access, the reader provides properties which allow reading all slots and all plugs in the instance store with a global iterator. For example, the iterator `Slots` contains all slot definitions of all contracts in the type store.

The methods `GetExtension`, `GetSlot`, and `GetPlug` allow an extension to access their meta objects by providing their *this* reference. Typically, this is convenient in methods which handle composer events (see usage example on page 102).

The methods `IsRegistered`, `IsPlugged`, and `IsSelected` allow the composition core to check wether a relationship exists between two objects in the meta model.

4.7.2 Instance Store Observable Interface

The instance store observable interface allows other components, such as composition tools or host extensions in an application, to register event listeners. The instance store notifies registered observers after an extension was added or removed, or after a slot was opened or closed, or after a relationship was added or removed.

```
interface InstanceStoreObservable {
  event ExtensionEventHandler ExtensionAdded;
  event ExtensionEventHandler ExtensionRemoved;
  event SlotEventHandler SlotOpened;
  event SlotEventHandler SlotClosed;
  event RegisterEventHandler Registered;
  event RegisterEventHandler Deregistered;
  event PlugEventHandler Plugged;
  event PlugEventHandler Unplugged;
  event SelectionEventHandler Selected;
  event SelectionEventHandler Deselected;
}
```

The observers register event handler methods. When the instance store fires an event, it includes information about what has changed as an argument.

```
delegate ExtensionEventHandler(object s, ExtensionEventArgs args);
delegate SlotEventHandler(object s, SlotEventArgs args);
delegate RegisterEventHandler(object s, RegisterEventArgs args);
delegate PlugEventHandler(object s, PlugEventArgs args);
delegate SelectEventHandler(object s, SelectEventArgs args);

class ExtensionEventArgs : System.EventArgs {
  ExtensionInfo Extension { get; internal set; };
}
class SlotEventArgs : System.EventArgs {
  SlotInfo Slot { get; internal set; }
}
class RegisterEventArgs : System.EventArgs {
  SlotInfo Slot { get; internal set; }
  PlugTypeInfo PlugType { get; internal set; }
  object GetParam(string name) { /* not shown */ }
}
class PlugEventArgs : System.EventArgs {
  SlotInfo Slot { get; internal set; }
  PlugInfo Plug { get; internal set; }
  object GetParam(string name) { /* not shown */ }
  object Object { get; internal set; }
}
class SelectEventArgs : System.EventArgs {
  SlotInfo Slot { get; internal set; }
  PlugInfo Plug { get; internal set; }
}
```

4.7.3 Instance Store Modifier Interface

The instance store modifier interface allows the composition core to modify instances in the instance store. The `AddExtension` method adds an instance; the `RemoveExtension` method

removes an instance. The `OpenSlot` method opens a slot; the `CloseSlot` method closes a slot. The other methods add and remove *registered*, *plugged* or *selected* relationships.

```
interface InstanceStoreModifier {
  void AddExtension(ExtensionInfo extension);
  bool RemoveExtension(ExtensionInfo extension);
  void OpenSlot(SlotInfo slot);
  void CloseSlot(SlotInfo slot);
  void Register(SlotInfo slot, PlugTypeInfo plugType);
  bool Deregister(SlotInfo slot, PlugTypeInfo plugType);
  void Plug(SlotInfo slot, PlugInfo plug);
  bool Unplug(SlotInfo slot, PlugInfo plug);
  void Select(SlotInfo slot, PlugInfo plug);
  bool Deselect(SlotInfo slot, PlugInfo plug);
}
```

All methods which remove an extension or a relationship return a boolean. They return *true* if the element to be removed was found, and they return *false* if the element was not found.

After the instance store processed a method call, it notifies registered observers about the changes. The following code shows how this pattern is implemented in the instance store. In this example, the `Plug` method fires the `Plugged` event.

```
class InstanceStore : InstanceStoreModifier, ... {
  public void Plug(SlotInfo slot, PlugInfo plug) {
    ...
    StorePlugged(slot, plug);
    OnPlugged(new PlugEventArgs() { Slot = slot; Plug = plug; });
    ...
  }
  void OnPlugged(PlugEventArgs args) {
    if(Plugged != null) Plugged(this, args);
  }
  void StorePlugged(SlotInfo slot, PlugInfo plug) { /* not shown */ }
}
```

The same pattern applies to the other modifier methods and events. Table 10 shows the operations and the associated events:

Instance Modifier Method	Instance Observer Event	Event Args
AddExtension	ExtensionAdded	ExtensionEventArgs
RemoveExtension	ExtensionRemoved	ExtensionEventArgs
OpenSlot	SlotOpened	SlotEventArgs
CloseSlot	SlotClosed	SlotEventArgs
Register	Registered	RegisterEventArgs
Deregister	Deregistered	RegisterEventArgs
Plug	Plugged	PlugEventArgs
Unplug	Unplugged	PlugEventArgs
Select	Selected	SelectEventArgs
Deselect	Deselected	SelectEventArgs

Table 10. Instance store notifications

4.8 Composition Core

The composition core creates and releases extensions and creates and removes relationships between extensions. It reads types from the type store and stores instances and their relationships in the instance store.

4.8.1 Creator Interface

The creator interface allows other components, such as host extensions in an application or composition tools, to create and release extensions. It contains the operations specified in Section 3.7.2 (see page 40).

```
interface Creator {
  ExtensionInfo CreateSharedExtension(ExtensionTypeInfo type);
  ExtensionInfo GetSharedExtension(ExtensionTypeInfo type);
  ExtensionInfo GetSharedExtension(
      ExtensionTypeInfo type, bool createIfRequired);
  ExtensionInfo CreateUniqueExtension(ExtensionTypeInfo type);
  void Release(ExtensionInfo extension);
}
```

The *CreateSharedExtension* method creates a shared instance of an extension type. The *GetSharedExtension* method returns the shared instance for an extension type. The *GetSharedExtension* method with the *createIfRequired* parameter is a convenience method which uses the following pattern to create the shared extension on demand.

```
ExtensionInfo GetSharedExtension(
    ExtensionTypeInfo type, bool createIfRequired) {
  ExtensionInfo shared = GetSharedExtension(type);
  if(shared == null && createIfRequired)
    return CreateSharedExtension(type);
  else
    return shared;
}
```

The *CreateUniqueExtension* method creates a unique instance for an extension type. The *Release* method releases an extension.

4.8.2 Composer Interface

The composer interface allows other components, such as composition tools or host extensions, to manually compose applications by opening and closing slots, and by connecting instances. The composer interface provides the composition operations specified in Section 3.7.3 (see page 41).

```
interface Composer {
  void OpenSlot(SlotInfo slot);

  void Register(SlotInfo slot, PlugTypeInfo plugType);
  void RegisterAll(SlotInfo slot);
  void RegisterPlug(PlugTypeInfo plugType);
  void Register(ExtensionInfo host, ExtensionTypeInfo contributor);
  void RegisterPlugs(ExtensionTypeInfo contributor);
  void RegisterAll(ExtensionInfo host);
```

```
  void Plug(SlotInfo slot, PlugInfo plug);
  void CreateAndPlug(SlotInfo slot, PlugTypeInfo plugType);
  void PlugWhereRegistered(PlugInfo plug);
  void CreateAndPlugWhereRegistered(PlugTypeInfo plugType);
  void CreateAndPlugAllRegistered(SlotInfo slot);
  void Plug(ExtensionInfo host, ExtensionInfo contributor);
  void CreateAndPlug(ExtensionInfo host, ExtensionTypeInfo contributor);
  void PlugWhereRegistered(ExtensionInfo contributor);
  void CreateAndPlugWhereRegistered(ExtensionTypeInfo contributor);
  void CreateAndPlugAllRegistered(ExtensionInfo host);

  void Select(SlotInfo slot, PlugInfo plug);

  void Deselect(SlotInfo slot, PlugInfo plug);

  void Unplug(SlotInfo slot, PlugInfo plug);
  void UnplugWherePlugged(PlugInfo plug);
  void UnplugAll(SlotInfo slot);
  void Unplug(ExtensionInfo host, ExtensionInfo contributor);
  void UnplugWherePlugged(ExtensionInfo contributor);
  void UnplugAll(ExtensionInfo host);

  void Deregister(SlotInfo slot, PlugTypeInfo plugType);
  void DeregisterPlug(PlugTypeInfo plugType);
  void DeregisterAll(SlotInfo slot);
  void Deregister(ExtensionInfo host, ExtensionTypeInfo contributor);
  void DeregisterPlugs(ExtensionTypeInfo contributor);
  void DeregisterAll(ExtensionInfo host);

  void CloseSlot(SlotInfo slot);
}
```

4.8.3 Configurator Interface

The configurator interface allows composition tools to configure the composition core. It contains the settings for the composition service specified in Section 3.7.4 (see page 50)

```
interface Configurator {
  bool AutoOpen { get; set; }
  bool AutoRegister { get; set; }
  bool AutoPlug { get; set; }
  bool AutoSelect { get; set; }
  bool LazyLoad { get; set; }
  bool AutoRelease { get; set; }
}
```

4.8.4 Observing the Type Store

The composition core observes changes in the type store. After a plug-in has been added to the type store, the composition core registers the extensions from this plug-in. If a plug-in has been removed from the type store, the composition core deregisters the extensions in this plug-in. The composition core registers event handler methods in the type store. The composition core reacts when a contract is added or removed, or when a plug-in is added or removed.

```
TypeStoreObserver typeStore = GetTypeStore();
typeStore.ContractAdded += ContractAdded;
typeStore.ContractRemoved += ContractRemoved;
```

```
typeStore.PluginAdded += PluginAdded;
typeStore.PluginRemoved += PluginRemoved;
```

When a contract has been added to the type store, the composition core revisits all slots and plugs which are affected by the new slot definitions. It opens slots which can be qualified after the slot definition has been added. And it registers plugs which qualify after the slot definition has been added.

```
public void ContractAdded(object s, ContractEventArgs args) {
  if(AutoOpen)
    InstanceStoreReader instanceStore = GetInstanceStore();
    foreach(SlotInfo slot in instanceStore.Slots)
      if(slot.AutoOpen && !slot.IsOpen())
        foreach(SlotDefinition slotdef in args.Contract.SlotDefinitions)
          if(slotDef == slot.SlotDefinition)
            OpenSlot(slot);
  if(AutoRegister)
    TypeStoreReader typeStore = GetTypeStore();
    foreach(PlugTypeInfo plugType in typeStore.PlugTypes)
      foreach(SlotDefinition slotdef in args.Contract.SlotDefinitions)
        if(plugType.AutoRegister && slotDef == plugType.SlotDefinition)
          composer.RegisterPlug(plugType);
}
```

When a contract has been removed from the type store, the composition core closes all slots which use one of the removed slot definitions.

```
public void ContractRemoved(object s, ContractEventArgs args) {
  InstanceStoreReader instanceStore = GetInstanceStore();
  foreach(SlotDefinition slotdef in args.ContractInfo.SlotDefinitions)
    foreach(SlotInfo slot in instanceStore.Slots)
      if(slotdef == slot.SlotDefinition)
        CloseSlot(slot);
}
```

When a plug-in has been added to the type store, the composition core registers all extensions in the plug-in in all matching slots of other extensions.

```
public void PluginAdded(object s, PluginEventArgs args) {
  if(!AutoRegister) return;
  foreach(ExtensionTypeInfo type in args.Plugin.Extensions) {
    RegisterPlugs(type);
}
```

When a plug-in has been removed from the type store, the composition core deregisters all extensions in the plug-in from all slots in which they were registered.

```
public void PluginRemoved(object s, PluginEventArgs args) {
  foreach(ExtensionTypeInfo type in args.Plugin.Extensions)
    DeregisterPlugs(type);
}
```

4.8.5 The Core Extension

The CI with the cores described in this chapter is packaged as a core extension, which acts as a root for Plux.NET applications (see Fig. 22). The core extension does not have any plugs, but it has two slots where other extensions can contribute. The *Discovery* slot integrates dis-

coverer extensions (see page 79). The *Startup* slot integrates startup extensions of applications. Below are the interfaces for the discovery and startup slot:

```
[SlotDefinition("Discovery")]
interface Discoverer {
  void StartDiscovery(TypeBuilder builder);
  event DiscoveryEventHandler DiscoveryEvent;
}

[SlotDefinition("Startup")]
[ParamDefinition("ExecuteInMainThread", typeof(bool), true)]
interface Startup {
  void Run();
}
```

The Discovery slot requires the `Discoverer` interface and does not require any parameters. The startup slot requires the `Startup` interface and the parameter `ExecuteInMainThread` of type `bool` with `true` as default value. After a contributor has been plugged in the startup slot, the core extension calls the `Run` method provided by the contributor. The parameter defines whether the *Run* method is called in the thread of the CI, or in a separate thread. The slot definitions are packaged in a contract *Plux.dll*.

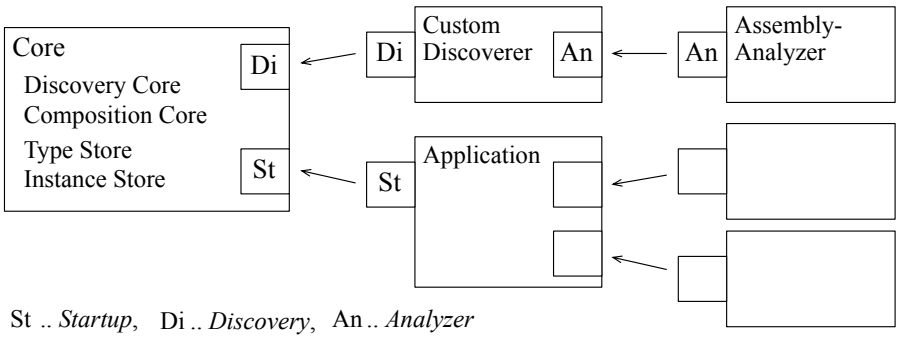


Figure 22. Core extension with discovery and startup slot

At startup, the composition core creates a unique core extension and opens its slots. Therefore it requires the composer, the creator, a modifier for the instance store, and the core contract.

```
InstanceStoreModifier instanceStore = GetInstanceStore();
ExtensionInfo core = creator.CreateUniqueExtension(coreType);
instanceStore.AddExtension(core);
foreach(SlotInfo slot in core.Slots) OpenSlot(slot);
```

The `CoreExtension` class implements the host behavior for the startup and discovery slot. After a startup contributor is plugged, the core extension retrieves the `ExecuteInMainThread` parameter. Depending on that parameter value, the core extension calls the `Run` method from the composition thread, or it creates a new thread for the contributor. In the case where the core creates a new thread, it also registers a handler for the *Released* event of the extension, because when the extension is released the core needs to abort the corresponding thread.

```
[Extension("Core")]
[Slot("Discovery", Multiple=true, Unique=true,
  OnPlugged="Discovery_Plugged", OnUnplugged="Discovery_Unplugged")]
[Slot("Startup", Multiple=true, Unique=true,
  OnPlugged="Startup_Plugged")]
```

```
class CoreExtension {
  var threadExtensions = new Dictionary<ExtensionInfo, Thread>();

  public void Startup_Plugged(object s, PlugEventArgs args) {
    bool mainThread = (bool) args.GetParam("ExecuteInMainThread");
    var startup = (Startup) args.Object;
    if(mainThread) startup.Run();
    else {
      Thread t = new Thread(startup.Run);
      args.Plug.Extension.Released += ThreadExtension_Released;
      threadExtensions.Add(args.Plug.Extension, t);
      t.Start();
    }
  }

  void ThreadExtension_Released(object s, ExtensionEventArgs args) {
    /* not shown */
  }
  ...
}
```

The discovery event handlers connect the discovery slot using the discovery registrar. After a discoverer has been plugged, the core extension registers it in the discovery core. After a discoverer has been unplugged, the core unregisters the discoverer from the discovery core.

```
DiscoveryCoreRegistrar registrar = GetDiscoveryCore(); /* not shown */

public void Discovery_Plugged(object s, PlugEventArgs args) {
  registrar.Register((Discoverer) args.Object);
}

public void Discovery_Unplugged(object s, PlugEventArgs args) {
  registrar.Unregister((Discoverer) args.Object);
}
```

Chapter 5: Plux.NET Applications

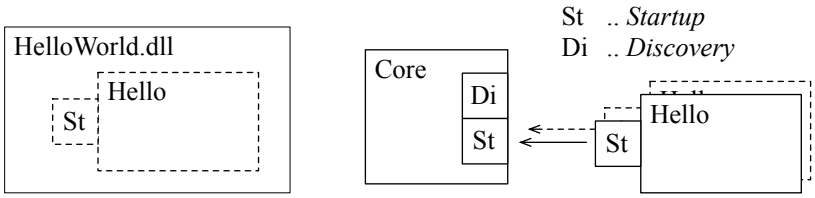
This chapter describes how to design and implement applications with the Plux.NET application programming interface (API). The API is universally applicable and can be used for any kind of .NET application. The Plux.NET API enables rich client applications which are customizable and dynamically reconfigurable. A customizable application allows users to load only components that they need for their current work thus keeping the application small and simple. An dynamically reconfigurable application can be reconfigured on the fly for different usage scenarios by dynamically swapping sets of plug-in components.

The Plux.NET framework comprises the runtime core, a framework library for rich client applications, and tools for composition and visualization. The runtime core implements the composition infrastructure described in Chapter 4. The runtime core composes an application from a component repository by plugging plugs of contributors into slots of the core or slots of other hosts. The framework library contains reusable extensions and a class library which simplifies building rich client applications that can be reconfigured dynamically. The tools visualize a composition, compose an application interactively or compose an application from a script.

This chapter is structured as follows: Section 5.1 describes how to create a Plux.NET extension. Sections 5.2 and 5.3 describe how to create a host extension using a slot. Sections 5.4 to 5.6 cover user interfaces which can change dynamically. Section 5.4 describes best practices. Section 5.5 describes how slot-bound widgets simplify the implementation of dynamic user interfaces. Section 5.6 presents a cross-compiler IDE as a case study.

5.1 Creating Startup Extensions

The "Hello World" example in this section shows how to create a Plux.NET application. A Plux.NET application needs to provide a startup extension. The startup extension plugs into the startup slot of the runtime core. Fig. 23a) shows the extension *HelloWorld*. It has a startup plug and is packaged in the plug-in *HelloWorld.dll*. Fig. 23b) shows the composed "Hello World" application. The *HelloWorld* extension was created and plugged into the startup slot of the *Core* extension.



a) HelloWorld plug-in b) HelloWorld application

Figure 23. Startup contributor of "Hello World" application

The following source code shows the C# implementation of the `HelloWorld` extension. The extension specifies metadata with attributes: The `Extension` attribute tags the class `HelloWorld` as an extension. Since the *Name* argument is omitted, the assembly analyzer will use the class name "HelloWorld" as the name for the extension. The `Plug` attribute adds a plug for the startup slot. The startup slot requires the contributor to implement the `IStartup` interface. In the `Run` method, the application prints "Hello world" and shuts down the runtime core to quit the application. If we do not shut down the runtime core, the application will continue to run, for example, in order to wait for dynamically discovered plug-ins. The `Param` attribute specifies that the `Run` method should be executed in a thread of the runtime core.

```
using System;
using Plux;

[Extension]
[Plug("Startup")]
[Param("ExecuteInMainThread", true)]

public class HelloWorld : IStartup {
  public void Run() {
    Console.WriteLine("Hello world");
    Runtime.Shutdown();
  }
}
```

We can now save the source code into a file *HelloWorld.cs* and use the following statement to build the Plux.NET plugin *HelloWorld.dll* with the C#-Compiler.

```
csc /t:library /out:HelloWorld.dll /reference:Plux.dll HelloWorld.cs
```

The following command executes the Plux.NET runtime core launcher. Table 11 (on page 94) lists the components of the Plux.NET composition framework and describes their purpose. The `discover` command line argument configures the bootstrap discoverer to discover the `HelloWorld.dll` plug-in. The `verbosity` argument configures the runtime core to log discovery and composition operations with normal verbosity. Table 12 (on page 94) lists the command line options for the runtime core launcher.

```
plux.exe /discover:HelloWorld.dll /verbosity:normal
```

The log below shows the operations in the discovery core while it discovers contracts and plug-ins of the `Plux.dll` runtime core assembly. Then the discovery core discovers the *HelloWorld.dll* plug-in, because we specified the `HelloWorld.dll` assembly as a command line argument. Next, the composer creates the `HelloWorld` extension and plugs it into the startup

slot (see Fig. 23b). After the `HelloWorld` extension was plugged, the core extension executes the `Run` method and the text "Hello world" is printed.

```
>plux.exe /discover:HelloWorld.dll /verbosity:normal
Plux.NET Version 0.3.1295.1

Contract added: 1:"Plux.dll"
Plugin added: 1:"Plux.dll"
Plugin added: 2:"HelloWorld.dll"
Hello world
```

The logger in the runtime core allows us to observe the composition process in more detail, if we specify `detailed` as the verbosity level. The log below shows how the composition proceeds. In phase 1, the composition core opens the startup slot of the `Core` extension. The bootstrap discoverer adds the *HelloWorld.dll* plug-in to the type store.

```
>plux.exe /discover:HelloWorld.dll /verbosity:detailed

Contract discovered: "Plux.dll"
Contract added: 1:"Plux.dll"
Plugin discovered: "Plux.dll
Plugin added: 1:"Plux.dll"
Slot opening: 1:"Discovery" (Core)
Slot opened: 1:"Discovery" (Core)
1 Slot opening: 2:"Startup" (Core)
1 Slot opened: 2:"Startup" (Core)
Plugin discovered: "HelloWorld.dll"
Plugin added: 2:"HelloWorld.dll"
2 Extension type registering:  2:"Hello" >> 1:"Core" (Startup)
2 Extension type registered:  2:"Hello" >> 1:"Core" (Startup)
3 Extension plugging:  2:"Hello" >> 1:"Core" (Startup)
3 Extension plugged:  2:"Hello" >> 1:"Core" (Startup)
Hello world
Slot closing: 1:"Discovery" (Core)
Slot closed: 1:"Discovery" (Core)
4 Slot closing: 2:"Startup" (Core)
4 Extension type deregistering:  2:"Hello" << 1:"Core" (Startup)
4 Extension unplugging:  2:"Hello" << 1:"Core" (Startup)
4 Extension unplugged:  2:"Hello" << 1:"Core" (Startup)
4 Extension type deregistered:  2:"Hello" << 1:"Core" (Startup)
4 Slot closed: 2:"Startup" (Core)
```

In phase 2, the composition core searches the type store for startup plugs and finds our `HelloWorld` extension. It registers the `HelloWorld` extension in the startup slot and thus makes the contributor known to the `Core` host.

In phase 3, the composition core creates a unique instance of the `HelloWorld` extension, plugs it into the startup slot, and notifies the `Core` extension that the contributor can be used. The `Core` extension retrieves the value of the parameter *ExecuteInMainThread.* Because the *HelloWorld* extension specified the value *true*, the *Core* extension calls the *Run* method of the contributor in the main thread. Finally, the *HelloWorld* extension prints the text "Hello world". This finishes the application and the runtime core can be shut down.

In phase 4, the runtime core shuts down. The composition core closes the slots of the `Core` extension. When the startup slot closes, the composition core unplugs and deregisters all contributors. After the slots were closed, the runtime core terminates. Fig. 24 summarizes phase 1 to 4.

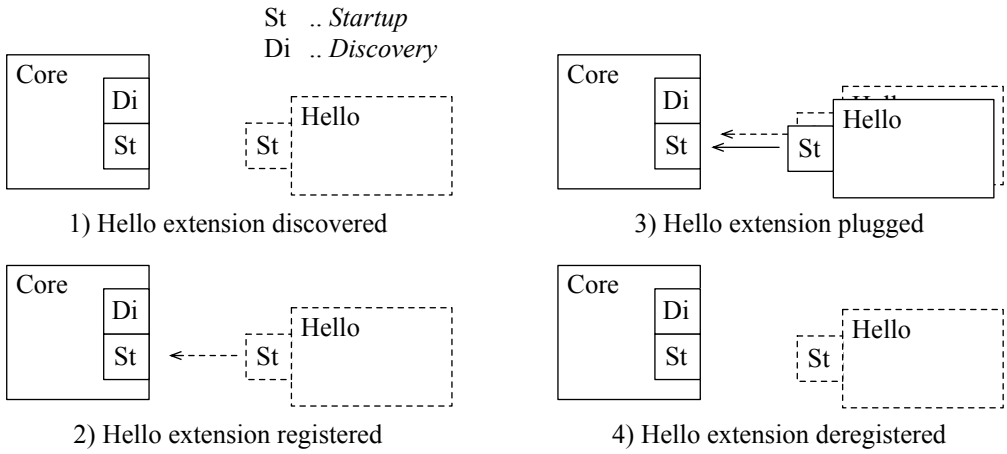


Figure 24. Composition process of "Hello World" application

File (Size)	Purpose
Plux.dll (214kB)	Composition infrastructure comprising the type store, instance store, discovery core, composition core, bootstrap discoverer, core contract, and core extension
Plux.Client.dll (22kB)	Event logging and diagnostics for rich client applications
Plux.Framework.dll (111kB)	Framework library for rich client applications
Plux.exe (24kB)	Runtime core launcher for rich client applications

Table 11. Components of the Plux.NET composition framework

Option	Purpose
/help	Display a usage message.
/nologo	Do not display the startup banner and copyright message.
/version	Display version information only.
/verbosity:<level>	Display this amount of information in the event log. The available verbosity levels are: quiet, minimal, normal, detailed, and diagnostic.
/discovery:<path>	Files and directories for bootstrap discovery of contracts and plug-ins. Default are all assemblies in the base directory of the application.
/base:<paths>	Directories which are registered in the ApplicationBase to discover plugins and contracts from there. Default is the base directory of the application.
/logfile:<path>	The Logger writes the messages in the specified file.

Table 12. Command line options of the Plux.NET runtime core launcher

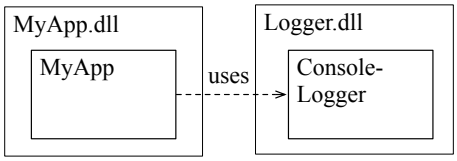
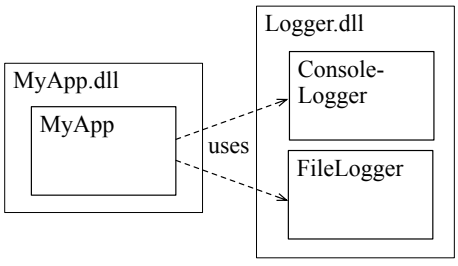
5.2 Creating Host Extensions Using Slots

A Plux.NET slot is the mechanism to make a host customizable and extensible. In the design of a Plux.NET application, a slot should be considered wherever one host component uses one or many other contributor components. But not every usage relationship should be designed with a slot. A slot is applicable, if one of the following two requirements holds: Firstly, a slot is applicable if contributors should be user-configurable, i.e., different users may use different contributors. Secondly, a slot is applicable if contributors should be dynamically changed, i.e., at run time, a contributor should be added, removed, or replaced.

Let us assume that we want to write an application that performs some actions and logs them. The application should use one or many loggers. For every action to be logged, the application will pass a log message including a time stamp to the logger.

Depending on customization and extensibility requirements, the composition scenarios as shown in Table 13 are conceivable. In scenario *a*, the application comprises one console logger and is deployed for every user in the same configuration. In scenario *b*, the application comprises two loggers which are used simultaneously. Again, the application deploys for every user in the same configuration. In neither scenario, it is intended to customize the loggers per user, or to change the logger at run time. Thus a slot is not applicable, hard-wired component usage is suitable.

In scenarios *c* to *f*, the logging should be flexible. We do not want to implement it hard-wired as part of the application, but rather as an extension. Therefore, we use a logger slot. In scenario *c*, each logger is packaged in a separate plug-in. The host application uses either one contributor if the slot has single cardinality, or all contributors if the slot has multiple cardinality. The application can be customized by deploying a different set of logger plug-ins. And the application can be extended through third party plug-ins.

Composition	Usage scenario
a) Use a hard-wired component...	...if your host component uses a *single* contributor, you deploy the same configuration for all users, and the configuration does not change at run time. 
b) Use multiple hard-wired components...	...if your host component uses *multiple* contributors, you deploy the same configuration for all users, and the configuration does not change at run time. 

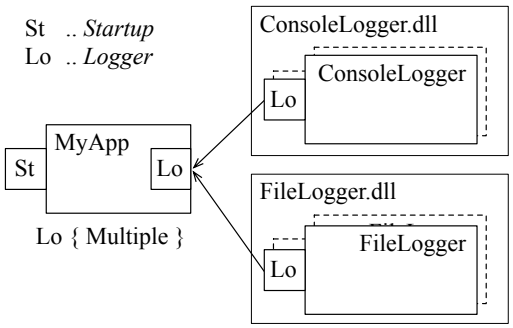
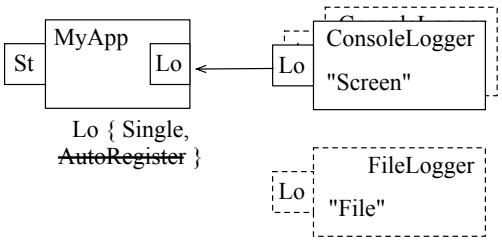
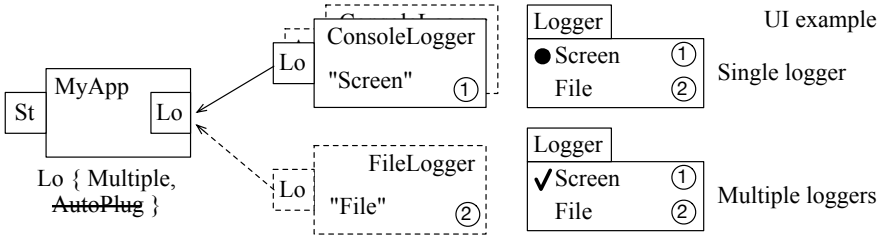
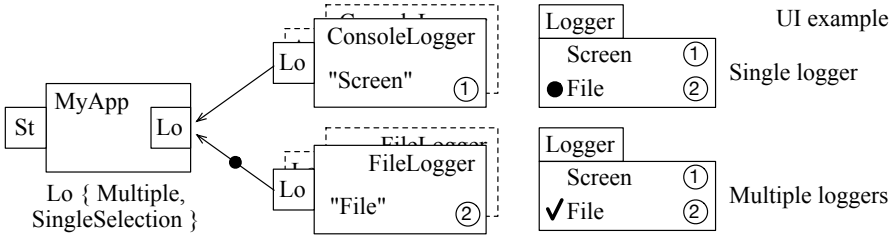
c) Use a single/multiple cardinality slot and package each contributor in a separate plug-in...	...if your host uses one/many contributors and contributors should be customizable per user (see page 99 for single cardinality sample code, or page 102 for multiple cardinality sample code). 
d) Use a single/multiple cardinality slot with manual registration...	...if your host uses one/many contributors which should be reconfigurable at run time. For example, to change the logger below, you would deregister the *ConsoleLogger* extension and register the *FileLogger* extension (see page 103 for sample code). 
e) Use a multiple cardinality slot with manual plugging...	...if your host uses one/many contributors and allows the user to choose contributors while the application runs (see page 105 for sample code).  e) Slot with manual plugging
f) Use a multiple cardinality slot with single/multiple selection ...	...if your host uses many contributors and allows the user to switch between multiple active contributors (see page 106 for sample code).  f) Slot with selection

Table 13. Slot composition scenarios

In scenario *d*, the logger slot is configured for manual registration. The configuration can be changed at run time by deregistering a logger and registering another logger to the slot. Typically, in this scenario the configuration change is not performed by the application itself, but rather by an external configuration tool. The configuration tool uses the composer interface of

the composition core to add or remove extensions. This scenario supports per-user configuration as scenario *c* did, but the configuration change is performed differently. In scenario *c* we add/remove a plug-in from/to a component repository. In scenario *d* we deploy all plug-ins to the component repository, and use a composition tool to manually register the desired extensions.

In scenarios *e* and *f*, the application allows the user to choose which contributor is used. In scenario *e*, the logger slot is configured for manual plugging. The application presents a list of the registered contributors to the user. For example, the application could retrieve a *Name* parameter from the contributor and create a menu item. When the user selects the menu item, the application plugs the contributor. When the user selects the same item again, the application unplugs the contributor.

While in scenario *e* contributors are unplugged and released, in scenario *f*, all contributors remain plugged and active. However, the host application uses only the selected contributor to log messages. In single selection slots, only one contributor can be selected at a time. In multiple selection slots, multiple contributors can be selected.

The following sections show how to use a slot according to scenarios *c* to *f*.

5.2.1 Specifying a Slot Definition

As a first step to make the host application customizable, we need to specify a slot definition. As we continue with the logger example, we define a slot for which logger extensions can contribute. The `ILogger` interface is tagged with a `SlotDefinition` attribute specifying "Logger" as the name of the slot. The logger slot has a parameter `Name` of type `string`, and a parameter `TimeFormat` of type `string`, which have to be provided by contributing extensions.

```
[SlotDefinition("Logger")]
[ParamDefinition("Name", typeof(string)]
[ParamDefinition("TimeFormat", typeof(string)]

public interface ILogger {
  void Print(string msg);
}
```

When designing a slot definition, a typical question is whether to use a parameter definition (as above), or a property in the interface (see below). The parameter definition is applicable, if the slot host retrieves the parameter when the contributor is registered. For example, when a host presents a list of registered contributors to the user, lets the user choose, and loads the chosen contributor lazily. The value of the parameter must be specified by the contributor at build time, if the Plux.NET attributes are used. Alternatively, with a custom discoverer, the parameter values could be provided at discovery time, by the discoverer.

The property in the interface is applicable, if the property value should be evaluated at run time, or if an application creates multiple instances of the same extension type, and the instances should provide different values.

```
[SlotDefinition("Logger")]
public interface ILogger {
  string Name { get; }
  string TimeFormat { get; }
  void Print(string msg);
}
```

In our example, we use the parameter definitions. We can now save the source code above into a file *ILogger.cs* and use the following statement to build the Plux.NET contract *MyApp.Contract.dll* with the C#-Compiler.

```
csc /t:library /out:MyApp.Contract.dll /reference:Plux.dll ILogger.cs
```

Although the discovery core of Plux.NET supports assemblies which contain extensions as well as slot definitions, we do not recommend this design. As a host extension uses a slot with the intention to be extensible, the creator of the host typically publishes the slot definition to third parties. We recommend to package slot definitions in a separate contract assembly and to append a *Contract* suffix to the assembly file name. For example, in our logger application the host plug-in is called *MyApp.dll* and the corresponding contract is called *MyApp.Contract.dll*.

The contract assembly serves as the collaboration interface between host and contributor. The creator of the host plug-in builds against the published contract assembly. The creator of the contributor plug-in builds against the contract assembly of his host. A direct build-time dependency between host and contributor plug-ins is not allowed. This design ensures that the runtime core can load the host plug-in without requiring that a specific contributor plug-in must be available, or load a contributor plug-in without requiring that a specific host plug-in must be available. The host and contributor plug-ins are independently loadable from each other, if the contract assembly is available, thus making host and contributor replaceable (see Fig. 25). Since the creator of the host wants that third parties contribute extensions for the host, he will not only publish the contract plug-in, but also the host plug-in, so that third parties can test against the host.

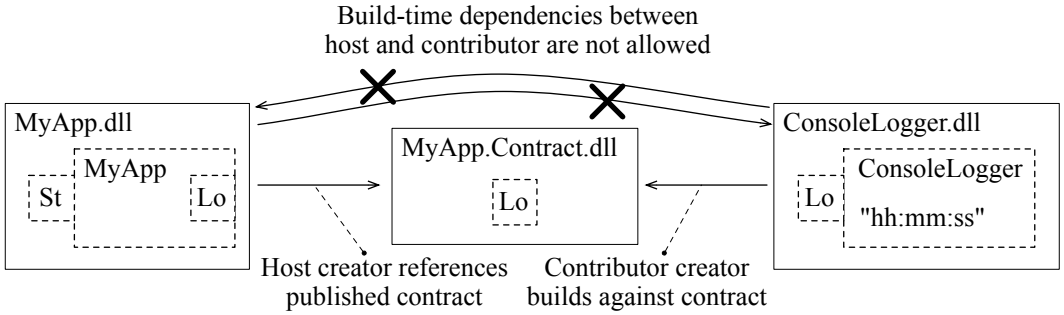


Figure 25. Build-time dependencies between contract and plug-in

A contract assembly can contain multiple related slot definitions. It should not contain extensions or other classes, for example, library classes. When a designer has to decide whether to package two slot definitions SD_1 and SD_2 in the same or in separate contract assemblies, two implications are relevant: Firstly, the slot definitions for a single host should not be distributed over multiple contracts. For convenience, the creator of a contributor should have to reference

only one contract when contributing to a single host. Secondly, as an exception to the first rule, if the interface used in the slot definition contains other types than those from the .NET base class library, the contract introduces an external build dependency. When the discovery core analyzes a contract assembly, it needs to resolve all external dependencies. If a dependency cannot be resolved, the contract fails to load. Thus, if two slot definitions SD_1 and SD_2 introduce different sets of external dependencies, they should be packaged separately.

5.2.2 Slot with a Single Contributor

The host application which uses the logger is itself an extension that plugs into the *Startup* slot of runtime core. So we need a `Plug` attribute for the startup slot and the application class needs to implement the interface `IStartup`. The `Param` attribute assigns `false` to the parameter *ExecuteInMainThread*, because the `Run` method should be executed in a separate thread. In the `Run` method, the host application performs some actions. As long as no logger extension is plugged, the host application performs the actions, but does not log the output.

```
[Extension]
[Plug("Startup")]
[Param("ExecuteInMainThread", false)]
[Slot("Logger", OnPlugged="Logger_Plugged", OnUnplugged="Logger_Unplugged")]
public class MyApp : IStartup {
  ILogger logger = null;
  string timeFormat;
  public void Run() {
    string msg;
    while(true) {
      DoSomeAction(out msg);
      if(logger != null)
        logger.Print(DateTime.Now.ToString(timeFormat) + ": " + msg);
      Thread.Sleep(1500);
    }
  }
  public void Logger_Plugged(object s, PlugEventArgs args) {
    timeFormat = (string) args.GetParam("TimeFormat");
    logger = (ILogger) args.Object;
  }
  public void Logger_Unplugged(object s, PlugEventArgs args) {
    logger = null;
  }
  void DoSomeAction(out string msg) { msg = "Hello"; }
}
```

The `Slot` attribute specifies that our application opens a logger slot. We handle the *Plugged* event in order to get notified when the composer plugs a logger extension. The event handler `Logger_Plugged` stores a reference to the plugged extension in the field `logger` and it stores the `TimeFormat` parameter in the field `timeFormat`. The `Run` method will use the plugged logger after it has been plugged, and it will stop using it, after it has been unplugged.

We can now save the source code above into file *MyApp.cs*. The following statement uses the C# compiler to build the host plug-in.

```
csc /t:library /out:MyApp.dll /reference:Plux.dll,MyApp.Contract.dll MyApp.cs
```

Creating Contributor Extensions

To complete the example, we write an extension that fits into the logger slot. The extension *ConsoleLogger* writes log message to the console window. The extension class `ConsoleLogger` is tagged with the `Extension` attribute and implements the `ILogger` interface. The `Plug` attribute tags the class as matching the `Logger` slot. The `Param` attributes assign values to the required parameters.

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

We can now save the source code above into a file *ConsoleLogger.cs*. The following statement uses the C# compiler to build the contributor plug-in.

```
csc /t:library /out:ConsoleLogger.dll /reference:Plux.dll,MyApp.Contract.dll
    ConsoleLogger.cs
```

The following command starts the application.

```
plux.exe /discovery:Logger.Contract.dll;MyApp.dll;ConsoleLogger.dll
```

The console logger produces the following output.

```
11:09:49: Hello
11:09:51: Hello
11:09:52: Hello
...
```

Fig. 26 shows the composed logger application.

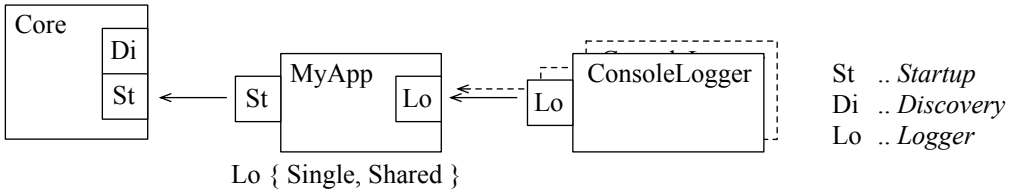


Figure 26. "Logger" sample application with single contributor

Customizing the Application

Since the host application uses a slot for the logger, the host can be customized. For example, by replacing the console logger with a file logger. We tag the `FileLogger` extension class with the extension and plug attribute, and we provide the values for the parameters. However, the file logger needs to open the log file when it is started, and close the log file when it is stopped. Therefore we provide event handlers for the *Created* and *Released* events.

```
[Extension(OnCreated="Logger_Created", OnReleased="Logger_Released")]
[Plug("Logger")]
[ParamValue("TimeFormat", "hh:mm:ss")]
```

```
public class FileLogger : ILogger {
  TextWriter stream = null;
  public void Logger_Created(object s, ExtensionEventArgs args) {
    stream = new StreamWriter("Logfile.txt");
  }
  public void Logger_Released(object s, ExtensionEventArgs args) {
    stream.Close();
  }
  public void Print(string msg) {
    stream.WriteLine(msg);
    stream.Flush();
  }
}
```

When we use composer events to initialize or clean up an extension, there are two candidates: We can use the *Created* and *Released* events of the extension (as above). This is applicable for contributors which are shared by multiple hosts. When the first host plugs the file logger, the logger is initialized. When the last host unplugs the file logger, the logger cleans up.

Alternatively, we can use the *Plugged* and *Unplugged* events of the plug (see below). Now, the behavior is different. Every time the logger is plugged, it creates a new log file. This is applicable for contributors of which hosts create unique instances, or when contributors need to reinitialize, after they have been replugged from one slot to another. In shared scenarios, this implementation is not applicable.

```
[Extension]
[Plug("Logger", OnPlugged="Logger_Plugged", OnUnplugged="Logger_Unplugged")]
[ParamValue("TimeFormat", "hh:mm:ss")]
public class FileLogger : ILogger {
  TextWriter stream = null;
  static int id = 0;
  public void Logger_Plugged(object s, PlugEventArgs args) {
    stream = new StreamWriter(String.Format("Logfile{0}.txt, ++id));
  }
  public void Logger_Unplugged(object s, PlugEventArgs args) {
    stream = null;
  }
  ...
}
```

For our example, we choose the variant with the *Created* and *Released* events. The following command starts the customized application with the new file logger plug-in.

```
plux.exe /discovery:Logger.Contract.dll;MyApp.dll;FileLogger.dll
```

So far, we have used the bootstrap discoverer. We specified on the command line which plug-ins we wanted to discover. Thus we had to restart the application every time we wanted to change the logger. The following section shows a custom discoverer which allows us to replace the logger without restarting the application. Thereby, the implementation of the host and the contributors remain unchanged.

Using a Custom Discoverer

The Plux.NET runtime core includes the bootstrap discoverer. If we want to discover plug-ins which are added to the component repository while the application is running, we need a custom discoverer. That custom discoverer can dynamically add plug-ins. The discovery core is

designed to integrate discoverer extensions (see page 79). Fig. 27 schematically shows how a custom discoverer, the directory watcher, watches a component repository folder and how the discoverer updates the type store upon changes: (a) After a file has been added to the folder, the directory watcher sends an *Add* notification to the discovery core. (b) After a file has been removed from the folder, the directory watcher sends a *Remove* notification to the discovery core.

The discovery core adds the discovered types to the type store, or removes types from the type store respectively. The composer observes the changes in the type store and recomposes the application accordingly.

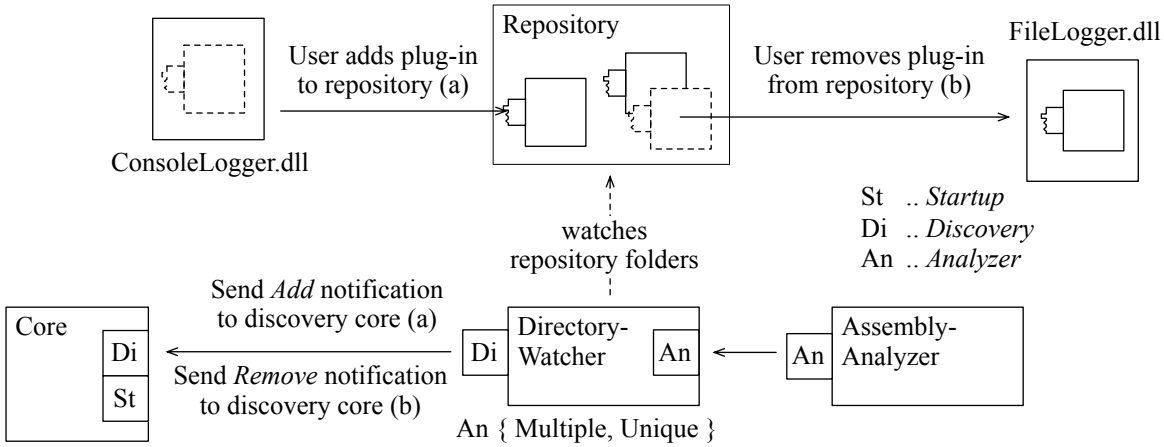


Figure 27. Dynamic custom discoverer extension "Directory Watcher"

Using the directory watcher, we can reconfigure the logger application without restarting it. The following command starts the application with the directory watcher.

```
plux.exe /discovery:Logger.Contract.dll;MyApp.dll;FileLogger.dll;
  DirectoryWatcher.dll
```

Let us assume we started the application as above with the file logger. We can now delete the *FileLogger.dll* from the application directory and thus remove it from the composition. If we subsequently add the *ConsoleLogger.dll* to the application directory, we successfully reconfigured the application without restarting it.

5.2.3 Slot with Multiple Contributors

Let us modify the logger example from the previous section for multiple simultaneous contributors. For every action to be logged, the application will pass a log message to all plugged contributors. Fig. 28 shows the modified host application which uses a console logger and a file logger.

The single contributor example from the previous section handled the *Plugged/Unplugged* events to maintain the reference to the contributor. We can modify this implementation and use a *List<ILogger>* collection to keep multiple contributor references. But we can also use a preferable technique, where we retrieve the plugged contributors from the composition model, instead of managing our own collection of contributors.

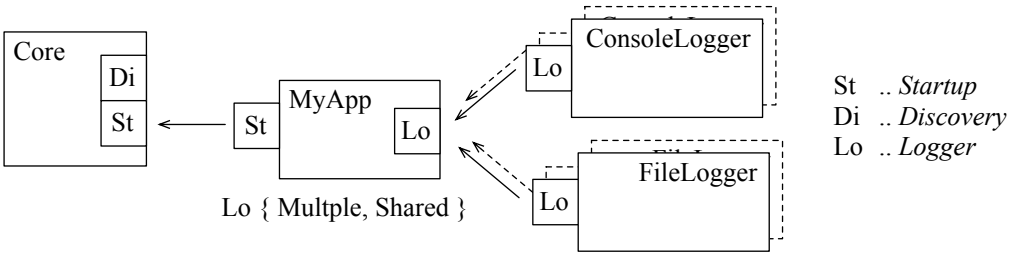


Figure 28. "Logger" sample application with multiple contributors

First, we set the multiple property of the slot attribute to *true*, thus directing the composer to register and plug multiple contributors. We omit the *Plugged* and *Unplugged* event handlers and access the composition model in the instance store instead. We obtain a reference to the logger slot and iterate over all plugged plugs. For each plugged contributor we retrieve the time format from the parameter and call the `Print` method.

```
...
[Slot("Logger", Multiple=true)]
public class MyApp : IStartup {
  public void Run() {
    string msg;
    while(true) {
      DoSomeAction(out msg);
      var slot = InstanceStore.GetSlot(this, "Logger");
      foreach(PlugInfo plug in slot.PluggedPlugs) {
        string timeFormat = (string) plug.Type.GetParam("TimeFormat");
        var logger = (ILogger) plug.Object;
        logger.Print(DateTime.Now.ToString(timeFormat) + ": " + msg);
      }
      Thread.Sleep(1500);
    }
  }
  void DoSomeAction(out string msg) { msg = "Hello"; }
}
```

The technique of reading plugged contributors from the instance store is applicable for single and for multiple cardinality slots. Thus we could rewrite the single contributor example from page 99 to use the technique shown here. This is even recommended, because using this implementation technique, changing from single to multiple cardinality is a matter of modifying the attribute.

The following command starts the application with both logger plug-ins.

```
plux.exe /discovery:Logger.Contract.dll;MyApp.dll;*Logger.dll
```

The example in this section showed how to implement a host application which uses a slot to integrate multiple logger contributors. If each logger extension is packaged in a separate plug-in, the application can be customized by adding, removing, or replacing logger plug-ins.

5.2.4 Manually Registering Contributors

In this section, we modify the logger example from the previous section to enable run-time reconfiguration using a composition tool. So far, we have used automatic registration, i.e., any

logger extension which was discovered, was automatically registered in the logger slot. To manually choose a logger, we set the slot to manual registration, and use a composition tool to manually register the loggers we want to start using, and deregister those which we want to stop using.

The modification in the host application is minimal. We disable automatic registration by setting the `AutoRegister` property to `false`. The rest of the host implementation remains unchanged. In our example, we configure the slot for single cardinality, but the example will also work with multiple cardinality.

```
[Slot("Logger", Multiple=false, AutoRegister=false)]
```

Initially, after we have started the application, the host will not use any logger at all, because no logger has been registered yet. A real-world application might provide a user-friendly composition tool to change the logger. For demonstration, we use the Plux.NET Console, a command interpreter extension included with the Plux.NET framework. The console allows the user to interactively compose an application by typing in composer commands.

```
plux.exe /discovery:Logger.Contract.dll;MyApp.dll;*Logger.dll;Console.dll
```

In the console window we can enter commands to browse the type store, to browse the instance store, or to control the composition core. For example, the *get-plugtype* command lists all plug types for the *Logger* slot in the type store. The pipe operator (`|`) uses the plug type objects as input for the next command. The next command retrieves the extension type for the plug type by reading the corresponding property. The resulting extension type objects are printed with their id, state, and name.

```
plux> get-plugtype | where Name=="Logger" | ExtensionType
  Id  State       ExtensionType
  10  DISCOVERED  ConsoleLogger
  11  DISCOVERED  FileLogger
  2 item(s).
```

The `register-extension` command uses the *Register(ExtensionInfo, ExtensionTypeInfo)* method from the composer interface (see page 86) to register the console logger. The composer then creates the logger and plugs it. The host application starts using the console logger.

```
plux> register-extension -Name ConsoleLogger
```

When we want to reconfigure the application, we use the `deregister-extension` command to remove the console logger, and the `register-extension` command to add the file logger.

```
plux> deregister-extension -Name ConsoleLogger
plux> register-extension -Name FileLogger
```

Fig. 29 shows the resulting application with the file logger in use. The console logger is not registered any more.

We have seen how to *use a composition tool* to reconfigure an application at run time. Registration makes a contributor known to a host, but a host does not know contributors which are not registered. A composition tool, such as the Plux.NET Console, can browse the type store for available contributors. A user chooses which contributor he wants to use with the composition and selectively registers those contributors. The following section will show how to choose a contributor *using the host application* itself.

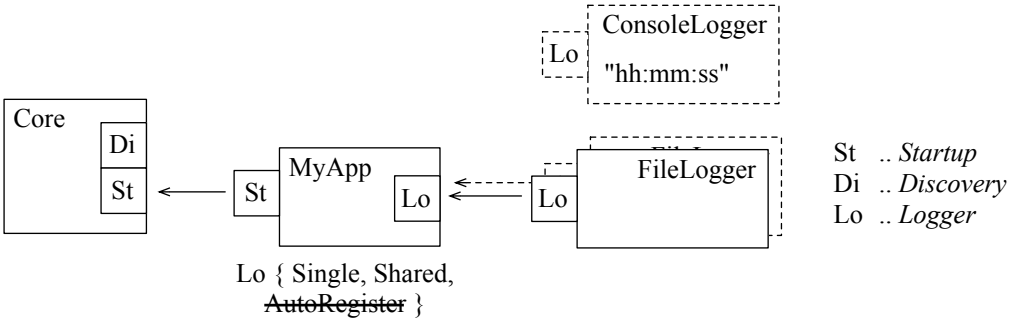


Figure 29. "Logger" sample application with manually registered contributor

5.2.5 Manually Plugging Contributors

In the previous scenario, the user used a composition tool to change the logger contributor. To choose a contributor in this way, means to reconfigure the application for a new working context. For example, a configuration for working context 1 might include the console logger, while the configuration for working context 2 includes the file logger.

In contrast, the scenario in this section, makes both logger contributors part of the same working context. The host extension provides UI, for example, a menu, which allows the user to choose which logger contributor should be used. So the user can switch back and forth between logger contributors. If the host allows the user to choose a contributor, i.e. displays which contributors are available and let the user pick the desired one, it configures the slot for multiple contributors, and disables automatic plugging.

Let us modify the logger example from the previous section in such a way that the host application allows the user to choose from available contributors. Fig. 30 shows the modified host application with multiple loggers registered, and only the file logger plugged.

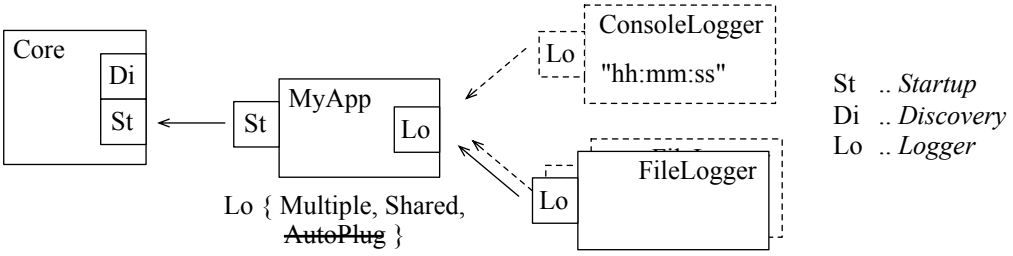


Figure 30. "Logger" sample application with manually plugged contributor

First, we set the `AutoPlug` property of the slot attribute to `false`, thus directing the composition core that we want to manually plug contributors.

```
[Slot("Logger", Multiple=true, AutoPlug=false)]
```

Next, we modify the `Run` method to present a menu. When the user presses any key, the host presents a menu which allows the user to choose a logger.

```
public void Run() {
    ...
```

```
    while(true) {
      if(Console.KeyAvailable) ShowMenu();
      DoSomeAction(out msg);
      ...
    }
  }
```

The menu displays a list of all registered loggers. For each logger, it displays the logger name retrieved from the parameter *Name* and a sequential number. Then the host waits for the user to enter a logger number, before it plugs the chosen contributor using the composition core.

```
void ShowMenu() {
  var slot = InstanceStore.GetSlot(this, "Logger");
  foreach(int i=0; i < slot.RegisteredPlugs.Count; i++) {
    PlugTypeInfo type = slot.RegisteredPlugs[i];
    Console.WriteLine("{0}: {1}, i, (string) type.GetParam("Name"));
  }
  int sel = Int32.Parse(Console.ReadLine());
  Composer.CreateAndPlug(slot, slot.RegisteredPlugs[sel]);
}
```

The example in this section has shown how to implement a host application which uses a slot to integrate multiple logger contributor extensions. The host application presents a menu with all available contributors, and lets the user choose a logger. The implementation in this section allows the user to plug multiple loggers, which are used simultaneously.

5.2.6 Manually Selecting Contributors

In this section, we modify the logger example from the previous section in such a way that the host application switches between the console and the file logger. In contrast to the previous section, the host application will send log messages not to all plugged contributors, but to the selected contributors only.

We enable automatic plugging and configure the slot for single selection, thus directing the composition core that we want all available loggers plugged, with one of them selected.

```
[Slot("Logger", Multiple=true, AutoPlug=true, SelectionMode=Single)]
```

We modify the menu so that it displays a list of all plugged loggers. The currently selected logger is marked with an asterisk. Then we select the chosen contributor using the composition core.

```
void ShowMenu() {
  var slot = InstanceStore.GetSlot(this, "Logger");
  foreach(int i=0; i < slot.PluggedPlugs.Count; i++) {
    PlugInfo plug = slot.PluggedPlugs[i];
    Console.WriteLine("{0}{1}: {2}",
        InstanceStore.IsSelected(slot, plug) ? "*" : " ", i,
        (string) plug.Type.GetParam("Name"));
  }
  int sel = Int32.Parse(Console.ReadLine());
  Composer.Select(slot, slot.PluggedPlugs[sel]);
}
```

To use only the selected contributor instead of all plugged contributors, we have to make one modification in the `Run` method. Instead of the `PluggedPlugs` property, we use the `SelectedPlugs` property.

```
public void Run() {
  string msg;
  while(true) {
    DoSomeAction(out msg);
    var slot = InstanceStore.GetSlot(this, "Logger");
    foreach(PlugInfo plug in slot.SelectedPlugs) {
      string timeFormat = (string) plug.Type.GetParam("TimeFormat");
      var logger = (ILogger) plug.Object;
      logger.Print(DateTime.Now.ToString(timeFormat) + ": " + msg);
    }
    Thread.Sleep(1500);
  }
}
```

Fig. 30 shows the modified host application with multiple loggers plugged, and the file logger selected.

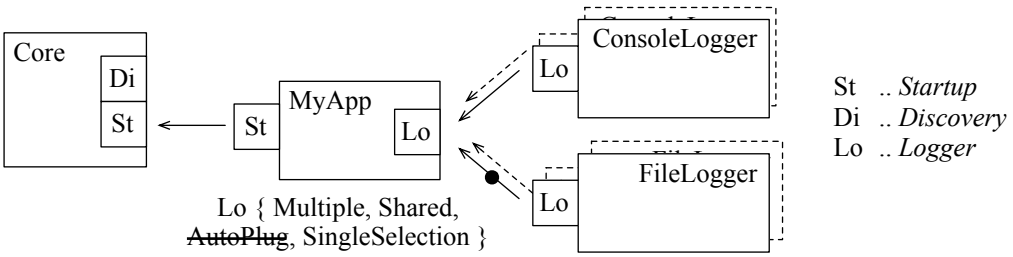


Figure 31. "Logger" sample application with manually selected contributor

If the slot is configured for single selection, we would not really need a for-loop for the selected contributors. However, if we use the loop, the implementation can be easily modified for multiple selected contributors, by changing the *SelectionMode* property of the slot to *Multiple*.

5.3 Shared, Unique, and Singleton Contributors

Section 5.2 showed how a host extension specifies the cardinality of a slot so that the slot uses either a single or multiple contributors. In contrast, sharing specifies whether a contributor should be used exclusively by one host or shared among multiple hosts. Applied to the logger example, that means multiple hosts can share one logger, or that each host uses a unique logger. The *Unique* property configures sharing, if set to *false*, the slot shares contributors. If the property is omitted, sharing is the default.

Whether a contributor is shared or unique is determined by the slot host. Another concept that affects sharing, the singleton, is specified on the contributor side. If a logger is marked as singleton, it can be shared among slots, however it cannot be used by more than one unique slots.

5.3.1 Sharing Contributors

If hosts use the same slot definition, they can share a contributor. Let us extend the logger example from the previous section so that we have multiple hosts. The slot definition, host extensions, and logger contributors are the same as in Section 5.2.2 (see page 99). The difference is that we now have two host extensions, and that both host extensions use a shared logger slot as shown in Fig. 32.

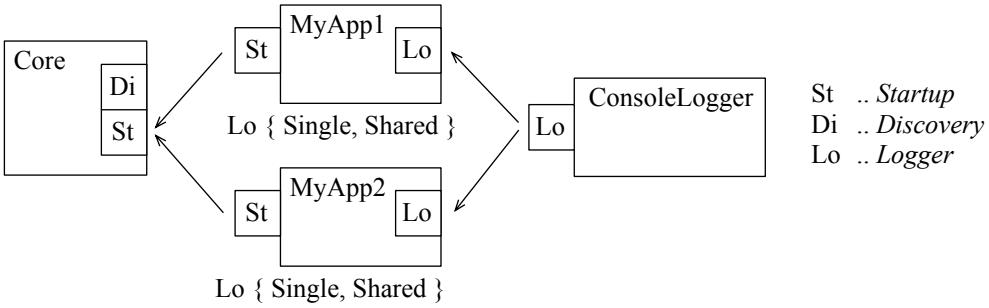


Figure 32. "Logger" sample application with shared contributor

To share a contributor, we set the `Unique` property to `false` in both hosts. These settings will direct the composition core to create a shared instance and plug it into both hosts.

```
[Extension]
[Slot("Logger", Unique=false)]
public class MyApp1 : IStartup { ... }

[Extension]
[Slot("Logger", Unique=false)]
public class MyApp2 : IStartup { ... }
```

Shared contributors are often combined with lazy loading. If the slots load their contributors lazily, the contributor is not plugged when it is registered. The instantiation is deferred until one of the hosts actually starts using the contributor. The implementations of the host application throughout Section 5.2 did not load lazily, because the logger was used in the *Run* method and thus was required immediately anyway. However, if the logging occurs in response to, for example, user interaction, loading the contributor lazily can speed up the application start.

We configure the slot for lazy loading by setting the `LazyLoad` property to `true`. Let us assume that we want to log a message when the user selects a menu item. In the handler for the *Clicked* event, we retrieve the registered contributors from the instance store. Then we use the *Creator* interface of the composition core to retrieve the shared instance of the contributor. If we pass `true` as second argument to `GetSharedExtension`, the composition core creates the shared instance on demand.

```
[Slot("Logger", Unique=false, LazyLoad=true)]

void MenuItem_Clicked(object s, EventArgs args) {
  var slot = InstanceStore.GetSlot(this, "Logger");
  foreach(PlugTypeInfo p in slot.RegisteredPlugs) {
    ExtensionInfo e = Creator.GetSharedExtension(p.ExtensionType, true);
    string timeFormat = (string) p.GetParam("TimeFormat");
    var logger = (ILogger) e.Object;
```

```
    logger.Print(DateTime.Now.ToString(timeFormat) + ": " + msg);
  }
  ...
}
```

When the composition core creates a shared instance, it automatically plugs that contributor into all slots which are configured for sharing and where the contributor is registered. In the example above, both hosts use a shared slot, thus the composition core plugs the logger into both hosts.

5.3.2 Unique Contributors

The alternate concept to contributor sharing are unique contributors. Even though multiple hosts use the same slot definition, they can require unique contributors instead of sharing a contributor. Fig. 33 shows two application hosts using unique instances of a contributor.

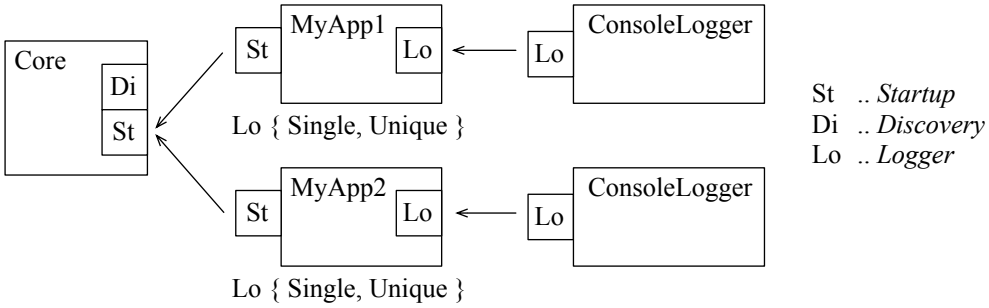


Figure 33. "Logger" sample application with unique contributors

To direct the composition core to create unique instances for both hosts, we set the `Unique` property to `true` in both hosts.

```
[Extension]
[Slot("Logger", Unique=true)]
public class MyApp1 : IStartup { ... }
```

```
[Extension]
[Slot("Logger", Unique=true)]
public class MyApp2 : IStartup { ... }
```

Like shared slots, unique slots can be combined with lazy loading. However, the implementation for loading unique contributors lazily is different than with a shared contributor, because each slot host has to create its own unique contributor. Applied to the scenario where a user clicks a menu item, the host calls the `CreateAndPlugAllRegistered` method from the `Composer` interface to plug all contributors.

```
[Slot("Logger", Unique=true, LazyLoad=true)]
```

```
void MenuItem_Clicked(object s, EventArgs args) {
  var slot = InstanceStore.GetSlot(this, "Logger");
  Composer.CreateAndPlugAllRegistered(slot);
  foreach(PlugInfo p in slot.PluggedPlugs) {
    string timeFormat = (string) p.Type.GetParam("TimeFormat");
    var logger = (ILogger) p.Object;
    logger.Print(DateTime.Now.ToString(timeFormat) + ": " + msg);
  }
}
```

This technique can also be applied for shared slots as an equivalent alternative to the implementation shown in Section 5.3.1 (see page 108). The technique can be applied for single as well as for multiple cardinality slots.

5.3.3 Singleton Contributors

In slots which are configured for unique contributors, the composition core expects contributors to be capable of providing multiple extension instances. Extensions might be implemented in a way that multiple instances are impeded, for example, if a contributor uses limited system resources. For example, our file logger implementation from Section 5.2.2 (see page 99) uses a single log file. This will cause problems if multiple hosts create unique instances of this contributor, because all instances would write to the same file.

To direct the composition core that it should not create multiple instances, we set the `Singleton` property to `true`.

```
[Extension(Singleton=true, OnCreated="Logger_Created", ...)]
[Plug("Logger")]
[ParamValue("TimeFormat", "hh:mm:ss")]
public class FileLogger : ILogger {
  public void Logger_Created(object s, ExtensionEventArgs args) {
    stream = new StreamWriter("Logfile.txt");
  }
  ...
}
```

Fig. 34 shows the composition result for the contributor above with the two host application from Section 5.3.1 (see page 108).

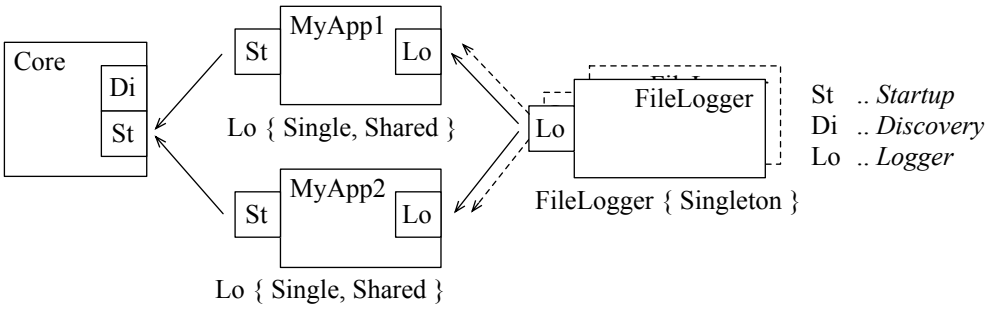


Figure 34. "Logger" sample application with shared singleton contributor

In shared slots, singleton contributors do not cause problems. However, if two hosts in an application use two unique slots, or already if one of the two slots is unique, the composition core cannot fill both slots. The composition core plugs the singleton in the first slot which opens, and will issue a warning message on all subsequent slots (see Fig. 35).

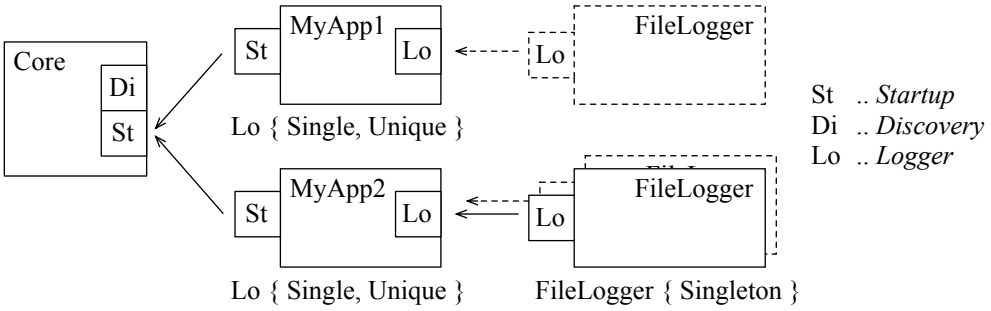


Figure 35. "Logger" sample application with unqiue singleton contributor

5.4 Best Practices for Dynamic User Interface Design

The benefit of a slot-based design is that applications can be customized. For rich client applications, this implies that the user interface (UI) must adapt when the configuration changes. Moreover, if the application can be reconfigured while it is running, the UI must change dynamically. In this section we show best practices for user interface design which consider that the UI will be adaptable at run time.

The examples in this section are based on the Windows Forms library, because Forms has been the standard UI library in .NET until version 3.0 (Sells and Weinhardt 2006). However, the underlying concepts discussed in this section can also be transferred to the Windows Presentation Framework introduced with Microsoft .NET 3.0 or to any other UI library for that matter.

In the following, we show four best practices: (i) *The Action slot* shows how to make adaptable widgets for commands, such as menus, toolbars, or buttons. (ii) *The View slot* shows how to build an adaptable multiple document interface (MDI) application. (iii) *The Control slot* shows how to build an adaptable composite user control using dynamic arrangement. (iv) *The DataSource slot* shows how to share a common data source among controls, for example, in a composite user control.

5.4.1 The Action Slot

The intent of the *Action* slot is to encapsulate a request as a contributor extension, so that a host can issue the request without knowing anything about the operation being requested or the contributor receiving the request. For example, user interface widgets like buttons or menus can use actions to carry out a request in response to user input.

The slot definition for the Action slot declares an interface for executing actions. The interface includes an `Execute` method, which is called by the host to issue the request. The `IsEnabled` method is called by the host before it displays a widget for a contributor. The response of the contributor is used to enable or disable the widget. Before a host can call the `IsEnabled` method, the contributor must be plugged. Since this counteracts lazy loading, the mechanism can be disabled for menu items that want to be always enabled. For action contributors which set the *LazyLoad* parameter on the action plug to *true*, the host will not call

`IsEnabled`, and will always enable the menu item instead. The `Text` parameter serves as a caption for the widget.

```
[SlotDefinition("Plux.Action")]
[ParamDefinition("Text", typeof(string))]
[ParamDefinition("LazyLoad", typeof(bool), true)]

public interface IAction {
  void Execute(object sender, ActionEventArgs args);
  bool IsEnabled(object sender, ActionEventArgs args);
}

public class ActionEventArgs {
  public Plug { get; internal set; }
  public Slot { get; internal set; }
}
```

Applying the Action Slot

For example, the menu is a well suited widget for action extensions. The menu host opens a slot for multiple action contributors. Each item in the menu corresponds to a registered action plug (see Fig. 36a). The menu host creates a menu item when a contributor is registered. It loads the contributor lazily when the user selects a menu item (see Fig. 36b). After the contributor is plugged, the host issues the request by calling the *Execute* method of the contributor.

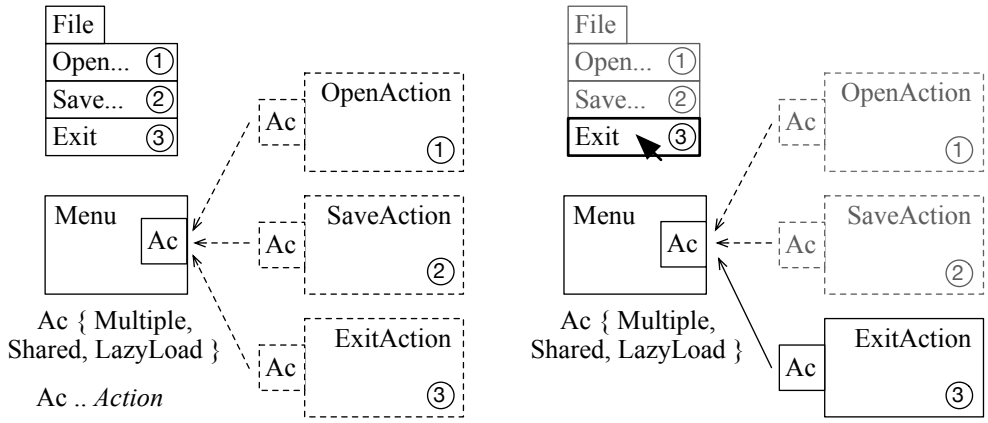


Figure 36. Best practice for the Action slot

After the contributor has executed the request, the host can unplug and release the contributor, because it is no longer needed. That keeps the application small. When a contributor is deregistered, the host removes the corresponding menu item.

The Action slot uses shared contributors, because it is intended that other hosts share the same contributors. For example, in rich client applications it is common, that a tool strip or a shortcut bar displays a subset of the items from the menu strip. In such an effort, a tool strip could open the same Action slot and register a subset of the action contributors.

The Action slot makes the menu extensible and customizable. The menu can be extended by providing a contributor for the Action slot. And the menu can be dynamically changed by reg-

istering the action contributors which are wanted, and by deregistering those which are not wanted.

Adjusting the Action Slot

The Action slot can be adjusted for different application scenarios:

- Set *AutoRelease=false* for the slot to avoid releasing of contributors after the request has been executed. This is applicable if the host wants to keep contributors alive, because the initialization of contributors takes a significant amount of time.
- Set *AutoRelease=false* for a contributor plug to avoid releasing of this contributor after its request has been executed. The reasoning is the same as above, however in this scenario, it is the initialization effort in that particular contributor which is non-negligible. Thus that particular contributor should not be released, while any other contributor should.
- Set *Unique=true* for the slot to prevent other hosts from sharing contributor instances with this slot. This can be useful if the contributors are stateful, and the state of the action contributor should not be shared among hosts.

Sample Code for the Host Extension

The `Menu` class below uses the Action slot. We configure the slot for multiple contributors, sharing and lazy loading. The menu class handles the *Registered/Deregistered* events and the *Plugged* event.

```
[Slot("Plux.Action", Unique=false, Multiple=true, LazyLoad=true,
  OnRegistered="Action_Registered", OnDeregistered="Action_Deregistered",
  OnPlugged="Action_Plugged")]
public class Menu : IControl {
  Control menu = new MenuStrip();
  ToolStripMenuItem fileMenu = new ToolStripMenuItem("File");
  public Menu() {
    fileMenu.DropDownOpened += DropDown_Opened;
    menu.Items.Add(fileMenu);
  }
  public void Action_Registered(object s, RegisterEventArgs args) { ... }
  public void Action_Deregistered(object s, RegisterEventArgs args) { ... }
  void DropDown_Opened(object s, EventArgs e) { ... }
  void MenuItem_Clicked(object s, EventArgs args) { ... }
  public void Action_Plugged(object s, PlugEventArgs args) { ... }
}
```

In Windows Form, all controls allow to store a reference to associated data in the `Tag` property. We use this property to store the plug type, when a contributor is registered, because we need the plug type when the user selects the menu item. The value of the `Text` parameter serves as a caption for the menu item. The `Controls` collection indexes items by their `Name` property. The `ToString` property returns a unique string representation for the plug type. We use that string as name for the menu item, because when the contributor will be deregistered, we use the same string to remove the menu item from the `Controls` collection. Finally, we register an event handler for the `Click` event, before we add the item to the menu.

```
public void Action_Registered(object s, RegisterEventArgs args) {
  var item = new ToolStripMenuItem {
    Tag = args.PlugType,
    Text = (string) args.GetParam("Text"),
    Name = args.PlugType.ToString() }
  item.Click += MenuItem_Clicked;
  menu.Controls.Add(item);
}
public void Action_Deregistered(object s, RegisterEventArgs args) {
  menu.Controls.RemoveByKey(args.PlugType.ToString());
}
```

When the user opens the file menu, the menu object receives the *DropDownOpened* notification. At this notification, the items in the submenu must be enabled or disabled. The first parameter contains the selected sub menu. For each menu item, we check the `LazyLoad` parameter. `True` means that menu item should be always enabled. Otherwise, we check if the contributor is already plugged. If not, we use the `Creator` to create a shared extension if necessary and call the `IsEnabled` method of the contributor to determine whether the menu item should be enabled.

```
void DropDown_Opened(object s, EventArgs args) {
  var subMenu = (ToolStripMenuItem) sender;
  var slot = InstanceStore.GetSlot(this, "Plux.Action");
  foreach(ToolStripItem i in subMenu.DropDownItems) {
    var plugType = (PlugTypeInfo) i.Tag;
    if((bool) plugType.GetParam("LazyLoad"))
      i.Enabled = true;
    else {
      Extension e = Creator.GetSharedExtension(plugType.Extension, true);
      var action = (IAction) e.Object;
      i.Enabled = action.IsEnabled(this,
        new ActionEventArgs { Slot = slot; Plug = e.Plugs[slot.Name]; });
    }
  }
}
```

When the user selects a menu item, the menu object receives a *Click* notification. The first parameter contains the selected menu item. We retrieve the plug type from the `Tag` property and plug the contributor using the composition core.

```
public void MenuItem_Clicked(object sender, EventArgs args) {
  var item = (ToolStripMenuItem) sender;
  var slot = InstanceStore.GetSlot(this, "Plux.Action");
  Composer.CreateAndPlug(slot, (PlugTypeInfo) item.Tag);
}
```

When the composition core sends the *Plugged* notification, the menu object retrieves the .NET object from the action contributor and calls the `Execute` method. As an argument, we pass the Action slot and the contributor plug. Library actions from the Plux.NET framework must work in any application, and therefore often make use of the meta objects (an example follows in the next section). Finally the command request is completed and we can unplug the contributor using the composition core.

```
public void Action_Plugged(object sender, PlugEventArgs args) {
  var action = (IAction) args.Object;
  action.Execute(this, new ActionEventArgs() {
```

```
        Slot = args.Slot, Plug = args.Plug });
    Composer.Unplug(args.Slot, args.Plug);
  }
```

Sample Code for the Contributor Extension

The `ExitAction` class is an extension and provides a plug for the Action slot. The `Param` attribute specifies "Exit" as text for the menu item. The `Execute` method calls a library action from the Plux.NET framework. This action exits the application by unplugging the startup extension of an application from the startup slot of the core extension. The exit action from the framework works in any Plux.NET application, because it uses the meta objects to find the right startup extension. Thereby it searches a path in the meta objects, from the action contributor plug up to the startup slot, and unplugs the startup extension into which the action is transitively plugged. This is the reason why the Action slot includes the *ActionEventArgs* object with slot and plug as parameter in the *Execute* method.

```
[Extension]
[Plug("Plux.Action")]
[Param("Text", "Exit")]
public class ExitAction : IAction {
  public void Execute(object sender, ActionEventArgs args) {
    Plux.Framework.ExitAction.Execute(sender, args);
  }
}
```

5.4.2 The View Slot

The intent of the *View* slot is to build an adaptable multiple document interface (MDI), so that a host integrates multiple child windows which reside under a single parent window. The host arranges the windows and manages their life-time, while the child windows are responsible for the window content. The View slot makes the MDI host customizable and extensible by turning the child windows into contributors.

The slot definition for the View slot specifies how the view host intends to integrate a view contributor. The interface includes a `Control` property, which is called by the host to get a Windows Forms-compliant control for the contributor. The host creates a child window and fills the client area of the window with the provided control. The host retrieves the `Name` property and uses it as a window caption. The slot definition also specifies a parameter called `Name`. The parameter is used by other host extensions, for example, by a menu if the view contributor is loaded lazily. The menu host needs to retrieve a caption for the menu item from the parameter when the contributor is registered. The `Name` property of the .NET object could not be accessed yet, because the contributor has not been instantiated.

```
[SlotDefinition("Plux.View")]
[ParamDefinition("Name", typeof(string))]
public interface IView {
  string Name { get; }
  Control Control { get; }
}
```

Applying the View Slot

Main windows of rich client applications can be implemented with view extensions. The main window host opens a slot for multiple view contributors. The view host ignores registration events. Instead, it creates a child window when a contributor is plugged and fills the window client area with the control supplied by the contributor (see Fig. 37). When a contributor is unplugged, the view host closes the associated child window. When a contributor is selected, the view host puts the input focus on the contributor. Other extensions, such as a menu, can use the selection to move the input focus between child windows.

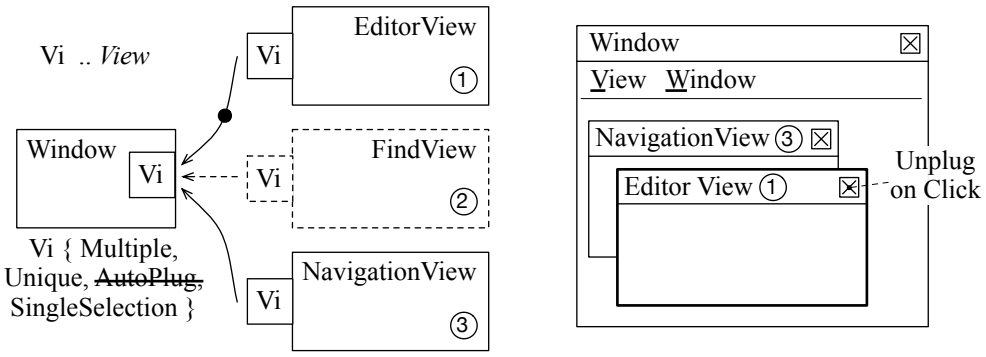


Figure 37. Best practice for the View slot

The window which hosts the view slot is typically combined with a menu. Menu items open or close a view, or move the input focus between views. Let us assume that the view host has a *View* and a *Window* menu as shown in Fig. 38. Although the two menus are independent and serve different purposes, they are coordinated through the view slot: The view menu can open or close a child window. For this purpose, it creates a menu item for each registered view contributor, and removes the menu item when a contributor is deregistered. To indicate which view is open, it checks menu items of plugged contributors. When the user selects an unchecked menu item, the menu creates and plugs the contributor. If a menu item was checked, i.e. the contributor was already plugged, the menu unplugs the contributor. The view host reacts to composer events accordingly. It opens a child window when a contributor is plugged, and closes a child window when a contributor is unplugged.

The window menu can move the input focus between child windows. For this purpose, it creates a menu item for each plugged contributor, and removes the menu item when a contributor is unplugged. To indicate which view currently has the focus, it checks the menu item of the selected contributor. When the user clicked a menu item, the menu selects the contributor. The view host reacts to the composer events accordingly and moves the input focus to the child window which corresponds to the contributor.

The View slot makes the main window extensible and customizable. The window can be extended by providing a contributor for the View slot. The main window and the associated menus can be customized by registering the view contributors which are wanted, and by deregistering those which are not wanted.

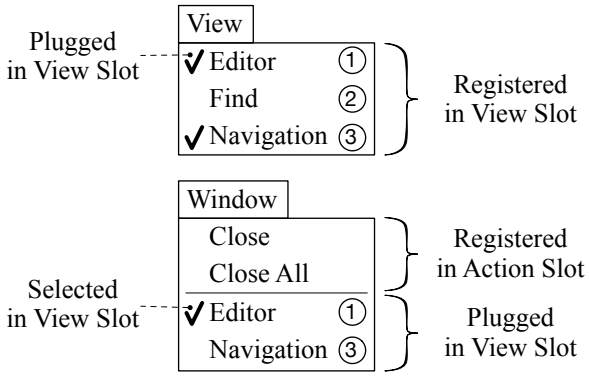


Figure 38. Dynamic menus for the View slot

Sample Code for the Host Extension

The `MainWindow` class below uses a View slot. We configure the slot for multiple unique contributors. Contributors for the view slot must be unique, because a view provides a control which is connected to an operating system widget. In Windows Forms, as in many other UI libraries, a widget cannot be part of multiple windows. This impedes sharing of extensions which provide a UI widget.

The main window creates a view contributor only when the user selects a menu item. For this reason, the host disables automatic plugging when contributors are registered. The main window class handles the *Plugged, Unplugged, Selected*, and *Deselected* events. The selection mode is `single`, because only one child window can have the input focus at a time. For the menu, the main window also handles the *Registered* and the *Deregistered* events.

```
[Slot("Plux.View",
  Unique=true, Multiple=true, AutoPlug=false, SelectionMode=Single,
  OnRegistered="View_Registered", OnDeregistered="View_Deregistered",
  OnPlugged="View_Plugged", OnUnplugged="View_Unplugged",
  OnSelected="View_Selected", OnDeselected="View_Deselected")]
public class MainWindow : IStartup {
  Form wnd = new Form();
  public void Run() { /* intentionally left blank */ }
  public void View_Registered(object sender, RegisterEventArgs args) { ... }
  public void View_Deregistered(object sender, RegisterEventArgs args) { ... }
  public void View_Plugged(object sender, PlugEventArgs args) { ... }
  public void View_Unplugged(object sender, PlugEventArgs args) { ... }
  public void View_Selected(object sender, SelectEventArgs args) { ... }
  public void View_Deselected(object sender, SelectEventArgs args) { ... }
  ...
}
```

When a contributor is plugged, we create a child window and store the plug in its `Tag` property, because we need the plug when the user closes the window (in Windows Forms a window is called *Form*). We retrieve the `Control` property from the view contributor and put the control on the client area of the child window. The `Name` property serves as a window title. Finally, we register event handlers for the *Activated* and *Closed* events of the child window, before we add the child window to the MDI children collection of the main window.

```
public void View_Plugged(object sender, PlugEventArgs args) {
  var view = (IView) args.Object;
  view.Control.Dock = DockStyle.Fill;
  var child = new Form() {
    Tag = args.PlugInfo,
    Text = (string) view.Name,
    MdiParent = wnd }
  child.Controls.Add(view.Control);
  child.Activated += Form_Activated;
  child.Closed += Form_Closed;
  child.Show();
  Form_Activated(child, null);
}
```

If the user closes a child window by clicking the widget in the window frame, the main window must react by unplugging the corresponding contributor: In the `Form_Closed` method we retrieve the plug from the `Tag` property of the closed window, and unplug the contributor using the composition core. However, the *Closed* notification also comes, when the child window is closed, because the slot was closed and the contributor was unplugged. We recognize this by checking the `IsReleased` property and do not unplug the contributor again, because this would cause a warning message from the composition core.

```
public void Form_Closed(object sender, EventArgs args) {
  var child = (Form) sender;
  var plug = (PlugInfo) child.Tag;
  if(plug.Extension.IsReleased) return;
  var slot = InstanceStore.GetSlot(this, "Plux.View");
  Composer.Unplug(slot, plug);
}
```

Let us assume that the user uses an action, for example a menu item, to close the window, instead of clicking the icon in the window frame. Then the process is reversed, i.e. the contributor is unplugged and the main window reacts by closing the form.

```
public void View_Unplugged(object sender, PlugEventArgs args) {
  foreach(Form child in wnd.MdiChildren)
    if(child.Tag == args.Plug) {
      child.Close();
      return;
    }
}
```

When child windows are opened and closed, the main window must keep the selected contributor and the input focus in sync. When the user has selected, for example, a menu item, to move the input focus to a child window, the menu item selects the corresponding contributor. The main window reacts to the *Selected* notification and sets the focus to the corresponding child window.

```
public void View_Selected(object sender, SelectEventArgs args) {
  foreach(Form child in wnd.MdiChildren)
    if(child.Tag == args.Plug) {
      child.BringToFront();
      return;
    }
}
```

The other way around, the user can move the input focus by clicking inside a child window. To handle that, the main window registered an event handler for the *Activated* event of the child form. After a child has been activated, the main window retrieves the plug from the `Tag` property of the child window, and selects the plug using the composition core.

```
public void Form_Activated(object sender, EventArgs args) {
  var child = (Form) sender;
  var slot = InstanceStore.GetSlot(this, "Plux.View");
  Composer.Select(slot, (PlugInfo) child.Tag);
}
```

Sample Code for the Contributor Extension

The `EditorView` class is an extension and provides a plug for the View slot. The `Param` attribute specifies "Editor" as a caption for a menu item, for example, in the view menu. To keep the example simple, we use a simplified implementation instead of a real editor here. The `Control` property returns a label with centered text. The `Name` property provides a document title to be displayed in the child window title.

```
[Extension]
[Plug("Plux.View")]
[Param("Name", "Editor")]
public class EditorView : IView {
  private readonly Label label = new Label();
  public EditorView() { label.Text = "Editor",
     label.TextAlign = ContentAlignment.MiddleCenter }
  public string Name { get { return "Document Title"; } }
  public Control Control { get { return label; } }
}
```

Sample Code for the View Menu

The main window handles the *Registered* and *Deregistered* events on the View slot to create and remove menu items. Thereby it uses the implementation as shown for the Action slot (see page 114).

```
ToolStripMenuItem viewMenu = new ToolStripMenuItem("View");
public void View_Registered(object s, RegisterEventArgs args) {
  var item = new ToolStripMenuItem {
    Tag = args.PlugType,
    Text = (string) args.GetParam("Text"),
    Name = args.PlugType.ToString() }
  item.Click += ViewMenuItem_Clicked;
  viewMenu.DropDownItems.Add(item);
}
public void View_Deregistered(object s, RegisterEventArgs args) {
  viewMenu.DropDownItems.RemoveByKey(args.PlugType.ToString());
}
```

When the user selects a menu item, the menu item sends the *Click* notification. We use the composition core to create and plug the registered contributor. As we have seen above, the main window handles the *Plugged* event where it displays a child window for the contributor.

```
public void ViewMenuItem_Clicked(object sender, EventArgs args) {
  var item = (ToolStripMenuItem) sender;
  var slot = InstanceStore.GetSlot(this, "Plux.View");
  Composer.CreateAndPlug(slot, (PlugTypeInfo) item.Tag);
}
```

The view menu should check menu items of plugged contributors. For that reason, we add code to the *Plugged* and *Unplugged* event handler methods from above. When a contributor is plugged, we check the menu item associated with the contributor. When a contributor is unplugged, we uncheck the menu item.

```
public void View_Plugged(object s, PlugEventArgs args) {
  ...
  viewMenu.DropDownItems[args.Plug.Type.ToString()].Checked = true;
}
public void View_Unplugged(object s, PlugEventArgs args) {
  ...
  viewMenu.DropDownItems[args.Plug.Type.ToString()].Checked = false;
}
```

Sample Code for the Window Menu

The Window menu is handled differently than the View menu. The purpose of the View menu is to create and plug contributors, whereas the purpose of the Window menu is to select a plugged contributor. To implement this behavior, we create a menu item, when the contributor is plugged, and remove it when the contributor is unplugged. Accordingly, we store a reference to the plug in the `Tag` property of the menu item, because we want to select the plug when the menu item is clicked.

```
ToolStripMenuItem windowMenu = new ToolStripMenuItem("Window");
public void View_Plugged(object s, RegisterEventArgs args) {
  ...
  var view = (IView) args.Object;
  var item = new ToolStripMenuItem {
    Tag = args.Plug,
    Text = view.Name,
    Name = args.Plug.ToString() }
  item.Click += WindowMenuItem_Clicked;
  windowMenu.DropDownItems.Add(item);
}
public void View_Unplugged(object s, RegisterEventArgs args) {
  ...
  viewMenu.DropDownItems.RemoveByKey(args.Plug.ToString());
}
```

When the user selects a menu item, we use the composition core to select the plug of the contributor. As we have seen above, the main window handles the *Selected* notification and brings the the child window of the selected contributor to the front.

```
public void WindowMenuItem_Clicked(object sender, EventArgs args) {
  var item = (ToolStripMenuItem) sender;
  var slot = InstanceStore.GetSlot(this, "Plux.View");
  Composer.Select(slot, (PlugInfo) item.Tag);
}
```

The window menu should check the menu item of the selected contributor. For that reason, we add code to the *Selected* event handler method from above, and add an event handler method for the *Deselected* event. When a contributor is selected, we check the menu item associated with the contributor. When a contributor is deselected, we uncheck the menu item.

```
public void View_Selected(object s, SelectEventArgs args) {
  ...
  windowMenu.DropDownItems[args.Plug.ToString()].Checked = true;
}
public void View_Deselected(object s, SelectEventArgs args) {
  windowMenu.DropDownItems[args.Plug.ToString()].Checked = false;
}
```

Actions in the Window Menu

The Window menu combines the View slot with the Action slot. It shows plugged view contributors, and additionally it handles the Action slot (see page 111). Using action contributors, the menu supports custom actions, for example, the *Close* and the *Close All* menu items below.

To integrate the action contributors, the main window host opens an *Action* slot and handles the *Registered* and *Deregistered* event (not shown). When an action contributor is registered, the main window retrieves the value of the `Text` parameter. It splits the string value at the ~ (tilde) character. The first part specifies in which sub menu the menu item belongs, the second part specifies the caption for the menu item.

The *Close* menu item in the window menu closes the focused child window. For the View slot, that means to unplug the selected contributor from the view slot. The `IsEnabled` property makes sure that the menu item cannot be clicked if no child window is open.

```
[Extension]
[Plug("Plux.Action")]
[Param("Text", "Window~Close")]
public class WindowCloseAction : IAction {
  public void Execute(object sender, ActionEventArgs args) {
    var viewHost = args.Plugs.PluggedInSlots["Plux.Action"];
    var viewSlot = viewHost.Slots["Plux.View"];
    Composer.Unplug(slot, slot.SelectedPlugs[0]);
  }
  public bool IsEnabled(object sender, ActionEventArgs args) {
    var viewHost = args.Plugs.PluggedInSlots["Plux.Action"];
    var viewSlot = viewHost.Slots["Plux.View"];
    return slot.SelectedPlugs.Count > 0;
  }
}
```

The *Close All* menu item in the window menu closes all open child windows. For the View slot that means to unplug all plugged contributors using the composition core.

```
[Extension]
[Plug("Plux.Action")]
[Param("Text", "Window~Close all")]
public class WindowCloseAllAction : IAction {
  public void Execute(object sender, ActionEventArgs args) {
    var slot = args.Plug.Extension.Slots["Plux.View"];
    Composer.UnplugAll(slot);
  }
  ...
}
```

5.4.3 The Control Slot

The intent of the Control slot is to encapsulate controls as contributors, so that a host can dynamically build a composite control. The host arranges the controls, while the child controls are responsible for the content. The Control slot makes the composite control customizable and extensible by turning the child control into a contributor.

The slot definition for the Control slot specifies how the slot host intends to integrate a control contributor. The interface includes a `Control` property, which is called by the host to get a Windows Forms-compliant control for the contributor. The host will arrange the control content on its client area. If the host arranges a single contributor, it will simply fill its client area with the contributor. If the host arranges multiple contributors, it will retrieve the `Position` parameter and arrange the contributors in the specified order. Simple host implementations could arrange the contributors either stacked or side-by-side. More sophisticated host implementations could use further parameters to arrange the contributors with more variations.

```
[SlotDefinition("Plux.Control")]
[ParamDefinition("Position", typeof(double)]
public interface IControl {
  Control Control { get; }
}
```

Applying the Control Slot

For example, view windows in rich client applications can be implemented with composite controls. The view host opens a slot for multiple control contributors. The view host ignores registration events. Instead, it puts the child controls of plugged contributors on its client area. The available client area space is divided among the contributors (see Fig. 39). When a contributor is plugged, the view host retrieves its position parameter, puts the child control of the contributor on the client area, and rearranges the other child controls according to the specified order. When a contributor is unplugged, the view host removes the child control of the contributor from the client area, and rearranges the other child controls in the specified order to fill the free space.

The Control slot makes the view extensible and customizable. The view can be extended by providing a contributor for the Control slot. Providers specify their desired position with a floating point number. The host will arrange the contributors accordingly. In the example from Fig. 39, a contributor could specify *Position=0.6* and would be arranged between the

structure tree and the find text box. The view can be customized by registering the child controls which are wanted, and by deregistering those which are not wanted.

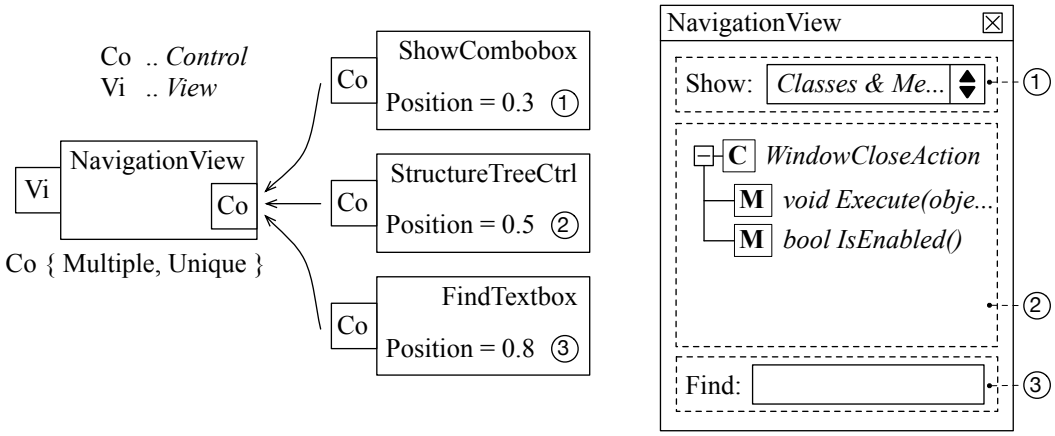


Figure 39. Best practice for the Control slot

Sample Code for the Host Extension

The `NavigationView` class is a contributor for the View slot. It implements interface `IView` and returns a flow layout panel from the `Control` property. In the constructor, we set the orientation for the flow layout panel as top-down. The flow layout panel will stack the controls from top to bottom in the order we add them.

The navigation view class uses the Control slot. We configure the slot for multiple unique contributors. The navigation view class handles the *Plugged* and *Unplugged* events.

```
[Extension]
[Plug("Plux.View")]
[Param("Name", "Navigation")]
[Slot("Plux.Control", Unique=true, Multiple=true,
  OnPlugged="Control_Plugged", OnUnplugged="Control_Unplugged")]
public class NavigationView : IView {
  FlowLayoutPanel panel;
  public NavigationView() {
    panel = new FlowLayoutPanel { FlowDirection = FlowDirection.TopDown; }
  }
  public Control Control { get { return panel; } }
  public void Control_Plugged(object s, PlugEventArgs args) { ... }
  public void Control_Unplugged(object s, PlugEventArgs args) { ... }
}
```

When a contributor is plugged we simply rearrange all controls. First, we add all plugged contributors to a `SortedList` using the retrieved `Position` parameter as a key. The list now contains the contributors in the correct display order. Then we iterate over the sorted list and add the controls to the flow layout panel. Since we use the unique `ToString` value as the key when we insert the control, we can remove the control using the same key when the contributor is unplugged.

```
public void Control_Plugged(object s, PlugEventArgs args) {
  panel.Controls.Clear();
  var list = new SortedList<double, PlugInfo>();
  foreach(PlugInfo p in args.Slot.PluggedPlugs)
    list.Add((double) p.GetParam("Position"), p);
```

```
  foreach(KeyValuePair<double, PlugInfo> item in list) {
    var c = (IControl) item.Value.Object;
    c.Control.Name = args.Plug.ToString();
    panel.Controls.Add(c.Control);
}
public void Control_Unplugged(object s, PlugEventArgs args) {
  panel.Controls.RemoveByKey(args.Plug.ToString());
}
```

Using the Control slot and an implementation like this, the navigation view is extensible and customizable. If the user registers a new control contributor, the view will dynamically re-arrange its controls.

Sample Code for the Contributor Extension

The `FindTextbox` class is an extension and provides a plug for the Control slot. The `Param` attribute specifies the value 0.8 as position, because we want to arrange the control below the structure tree in the navigation view (see Fig. 39 on page 123). The shown implementation is simplified and leaves out the actual find functionality. We use a left to right-oriented flow layout panel to arrange a label and a text box as shown in Fig. 39.

```
[Extension]
[Plug("Plux.Control")]
[Param("Position", 0.8f]
public class FindTextbox : IControl {
  FLowLayoutPanel panel = new FlowLayoutPanel();
  public FindTextbox() {
    panel = new FlowLayoutPanel {
      FlowDirection = FlowDirection.LeftToRight; }
    Control.Controls.Add(new Label { Text = "Find:" });
    Control.Controls.Add(new TextBox());
  }
  public Control Control { get { return panel; } }
}
```

5.4.4 The DataSource Slot

The intent of the *DataSource* slot is to share a data source among multiple extensions which are plugged into the same host. The motivation is, that if in a data-driven application an extensible host integrates third-party contributors into its slots, those contributors need access to the data. The host initializes the common data source as a shared extension. The contributors open a data source slot and share the data source extension with their host. The data source slot is a supplement to the View slot and the Control slot. With the data source slot, the extensible view host and its control contributors can share a common data source.

Data source contributors must provide two plugs: A *data source initializer* plug, which is used by the main host to set up the data source. And a *data source* plug, which is used by all hosts to access the data. The slot definition for the DataSource slot specifies how a host intends to access the data source. The interface includes a `Name` property, which is called by the host to display a title. The `Data` property returns a reference to the actual data. In a data-driven appli-

cation this might be a table-oriented data set, in a text editor application it might be a text buffer. The `Changed` event allows extensions to register for change notifications.

```
[SlotDefinition("Plux.DataSource")]
public interface IDataSource {
  string Name { get; }
  object Data { get; }
  event EventHandler Changed;
}
```

The slot definition for the DataSourceInit slot specifies how the main host initializes the data source. The interface includes an `Open` method, which is called by the host to connect to the data source. A data table source connects to the database, a file-based source opens the file. The `Load` method loads the data into memory. The `Save` method saves the data back to the data source. The `Close` method closes the data source. A data table source disconnects from the database, a file-based source closes the file.

```
[SlotDefinition("Plux.DataSourceInit")]
public interface IDataSourceInit {
  void Open(object source);
  void Load();
  void Save();
  void Close();
}
```

Applying the DataSource Slot

The section about the Control slot shows how View windows in rich client applications can be implemented with control extensions (see page 122). Typically, controls which belong to the same view host share data using the data source slot. The view host has a slot for controls and a slot for the data source. The control slot is opened manually. Initially it is closed, because the control contributors require an initialized data source to be functional (see Fig. 40a). Thus the view host waits until the data source is plugged, and then opens the control slot manually (see Fig. 40b).

After the data source was plugged and the control slot was opened, the control contributors are plugged and the UI is arranged. Each control contributor opens its shared data source slot and gets the same data source plugged as the view host is using. View and control contributors share a data source (see Fig. 40c). For example, the structure tree in the navigation view (compare Fig. 39 on page 123) parses classes and method signatures from the text buffer data source. If the content of the data source changes, the data source triggers an event and all controls update their content.

The data source can be exchanged without closing and reopening the view window. When the old data source is unplugged, the view host closes the control slot and releases all control contributors. When the new data source is plugged, the view host reopens the control slot and the controls use the new data source.

In the same way that the view host shares the data source with its control contributors, the window host shares the data source with its view contributors. The window host uses the data source initializer plug, because it initializes the data source, i.e. it opens the data source and loads the data into memory. For the text editor example, this means that the window host loads a file into the text buffer. Fig. 40d shows the window host, which opens the data source

initializer slot and configures the view slot for manual opening. When the data source plugs, the window host initializes the data source, and opens the view slot. This causes the views to open their control slots. All control contributors from all views share the common data source. In the example, the editor view allows editing the text in the buffer, the structure view shows an outline of the file, and the find view allows searching the text.

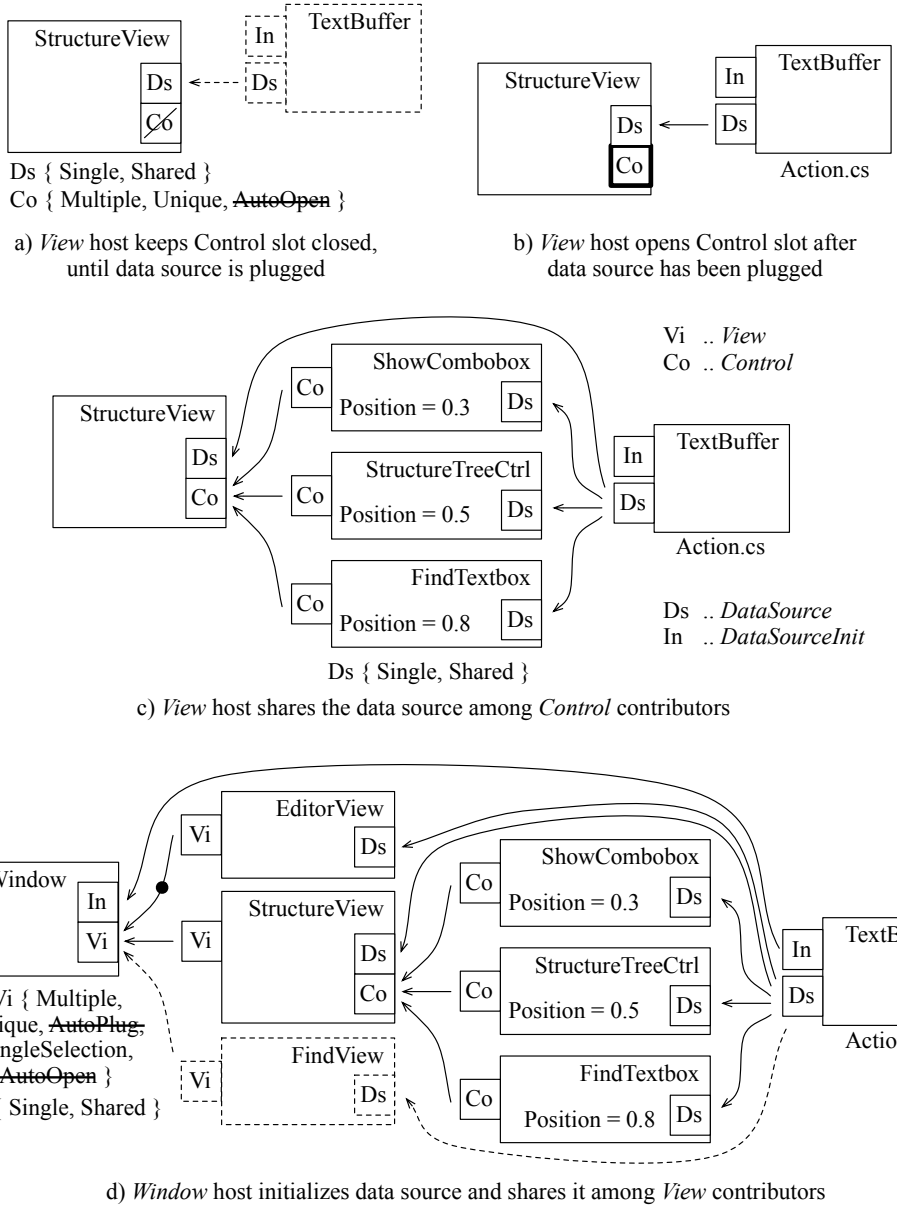


Figure 40. Best practice for the DataSource slot

Sample Code for the Host Extension

The `StructureView` class uses a DataSource slot. It handles the *Plugged* and *Unplugged* events. It also has a Control slot, which is configured for manual opening. When a data source contributor is plugged, the structure view class opens the control slot using the composition core. When the data source contributor is unplugged, it closes the control slot. This mechanism makes sure that controls are only created when the shared data source is ready.

```
[Extension]
[Slot("Plux.Control", AutoOpen=false, ...)]
[Slot("Plux.DataSource",
  OnPlugged="DataSource_Plugged", OnUnplugged="DataSource_Unplugged")]
public class StructureView : IView {
  public void DataSource_Plugged(object s, PlugEventArgs args) {
    var slot = InstanceStore.GetSlot(this, "Plux.Control");
    Composer.OpenSlot(slot);
  }
  public void DataSource_Unplugged(object s, PlugEventArgs args) {
    var slot = InstanceStore.GetSlot(this, "Plux.Control");
    Composer.CloseSlot(slot);
  }
  ...
}
```

The `MainWindow` class uses the same technique on the DataSourceInit slot. When the data source is plugged, the main window initializes the data source. It loads text from a file into the buffer, before it opens the view slot. Then the composition core plugs the views and plugs the controls into the views. The result is that all contributors share the same data source.

```
[Extension]
[Slot("Plux.DataSourceInit", OnPlugged="DataSourceInit_Plugged", ...)]
public class MainWindow : ... {
  public void DataSourceInit_Plugged(object s, PlugEventArgs args) {
    var source = (IDataSourceInit) args.Object;
    source.Open("Action.cs");
    source.Load();
    var slot = InstanceStore.GetSlot(this, "Plux.View");
    Composer.OpenSlot(slot);
  }
  ...
}
```

The `StatisticsLabel` class contributes to the Control slot. To keep the example simple, we use a simplified implementation. The statistics control opens a DataSource slot. When a data source is plugged, we register the `Update` method in the `Changed` event of the data source. Upon changes in the data source, the control displays the current size of the text buffer.

```
[Extension]
[Plug("Plux.Control")]
[Param("Position", 0.8f]
[Slot("Plux.DataSource", OnPlugged="DataSource_Plugged")]
class StatisticsLabel : IControl {
  Label label = new Label();
  Buffer buffer = null;
  public Control Control { get { return label; } }
  public void DataSource_Plugged(object s, PlugEventArgs args) {
    var source = (IDataSource) args.Object;
    buffer = (Buffer) source.Data;
    source.Changed += Update;
    Update();
  }
  void Update() {
    label.Text = String.Format("Size: {0}", buffer.Length);
  }
}
```

Sample Code for the Contributor Extension

The `TextBuffer` class is an extension and provides plugs for the DataSource and the DataSourceInit slot. In the `Load` method, the buffer class reads the text from a file into memory.

```
[Extension]
[Plug("Plux.DataSource")]
[Plug("Plux.DataSourceInit")]
class TextBuffer : IDataSource, IDataSourceInit {
  Buffer buf = new Buffer(); // class Buffer not shown
  string fileName;
  public object Data { return buf; }
  public void Open(object source) { fileName = source; }
  public void Load() {
    using(TextReader r = new StreamReader(fileName))
      buf = new Buffer(r.readToEnd());
    }
  }
  ...
}
```

5.5 Binding Widgets to Slots

The best practices in Section 5.4 showed how slots can be used to build a user interface which is dynamically reconfigurable. In the scenarios where the application allows the user to choose a contributor, we have used a menu widget to choose. The implementation of a widget that reacts to the events of the composition core requires some effort. Since much of the implementation is generally applicable, the Plux.NET framework comes with classes that bind slots to Windows Forms widgets. This section describes bindings for the following widgets: a menu, a listbox/combobox, a checked listbox, and a tab control. All widgets can be combined with a plug behavior or with a select behavior.

5.5.1 Widgets with Plug Behavior

Slot-bound widgets with plug behavior are applicable for host extensions which use a multiple cardinality slot and manual plugging. Such a host uses one or many contributors and allows the user to choose the contributor while the application runs (see scenario *e* in Section 5.2 on page 96).

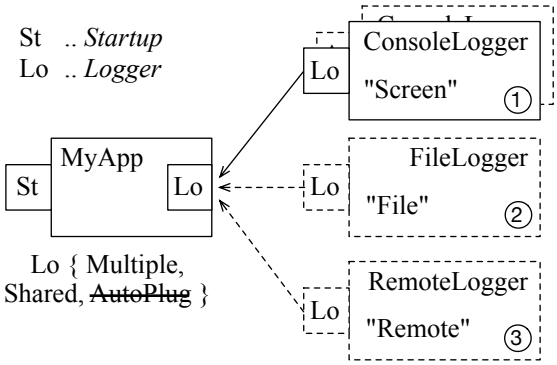


Figure 41. "Logger" sample application with plug behavior

Fig. 41 shows a host application which allows the user to choose which contributor should be used. Therefore, the host application can use a slot-bound widget with plug behavior. If a widget is bound to a slot, it registers as an observer in the instance store, and observes all composer events for this slot. For example, the following C# code in the host extension `MyApp` creates a combo box and binds it to the logger slot (see Fig. 42b).

```
[Extension(OnCreated="Extension_Created")]
public class MyApp : ... {
  ComboBox combo;
  public void Extension_Created(object s, ExtensionEventArgs args) {
    combo = new ComboBox();
    combo.Format += FormatItem;
    combo.BindWithSinglePlugBehavior(
      InstanceStore.GetSlot(this, "Logger"));
  }
  public void FormatItem(object s, ListControlConvertEventArgs args) {
    var plugType = (PlugTypeInfo) args.ListItem;
    args.Value = (string) plugType.GetParam("Name");
  }
  ...
}
```

The binding implementation uses C# extension methods to add the `BindWithSinglePlugBehavior` method to the existing Windows Forms *ComboBox* class. The single plug behavior creates a list item when a contributor is registered, and removes it when the contributor is deregistered. To control which text is used for the list item, we use Windows Forms format events (Sells and Weinhardt 2006). The `Format` event is raised, before each visible item in the combo box is formatted. Handling this event gives us access to the string to be displayed. In this example, we retrieve the `Name` parameter for the item. In a similar way the slot could be bound to a menu strip (Fig. 42a).

When the user changes the selection in the combo box, the binding uses the composition core to implement the single plug behavior. Single plug behavior means that only one contributor can be plugged at a time. Thus, the binding first unplugs all contributors, and then creates and plugs the chosen contributor using the composition core.

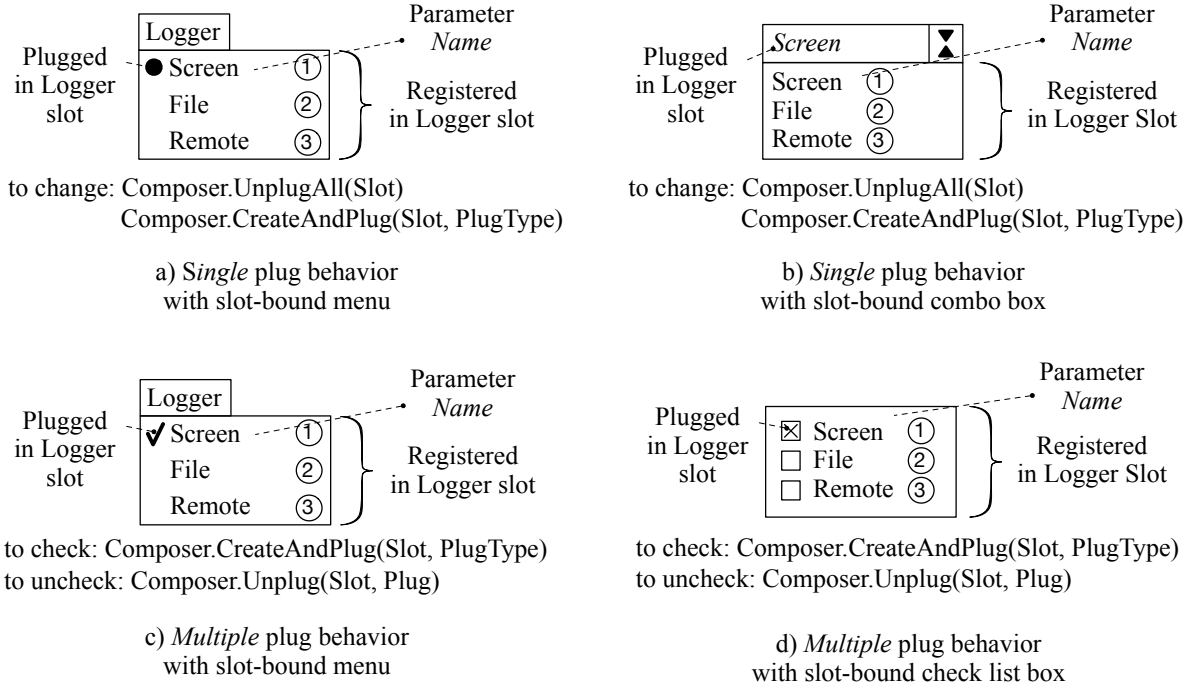


Figure 42. Slot-bound widgets for plug behavior

The extension method *BindWithMultiplePlugBehavior* binds a slot to a menu strip or a checked list box with multiple plug behavior (see Fig. 42c+d). The multiple plug behavior handles registration events in the same way as the single plug behavior. However, it allows multiple contributors to be plugged at the same time. Therefore, the binding creates and plugs a contributor if an unchecked item is clicked, and it unplugs a contributor if a checked item is clicked.

5.5.2 Widgets with Select Behavior

Slot-bound widgets with select behavior are applicable for hosts which use a multiple cardinality slot, automatic plugging, and single/multiple selection. Such a host uses one or many active contributors and allows the user to switch between the contributors while the application runs (see scenario *e* in Section 5.2 on page 96).

Fig. 43 shows a host application which allows the user to switch between contributors. Therefore, the host application can use a slot-bound widget with select behavior. The extension method *BindWithSingleSelectBehavior* binds a slot to a menu strip or a combo box (see Fig. 44a+b). The single select behavior creates a list item when a contributor is plugged, and removes it when the contributor is unplugged. The behavior checks the selected contributor. When the user changes the selection, the behavior uses the composition core to select the corresponding contributor. The composition core automatically deselects any other contributor, if the slot is configured for single selection.

The extension method *BindWithMultipleSelectBehavior* binds a slot to a menu strip or a checked list box with multiple select behavior (see Fig. 44c+d). The multiple select behavior handles plugging events in the same way as the single select behavior. However, it allows

multiple contributors to be selected at the same time. Therefore, the behavior uses the composer to select a contributor when the user checks an item, and to deselect the contributor when the user unchecks an item.

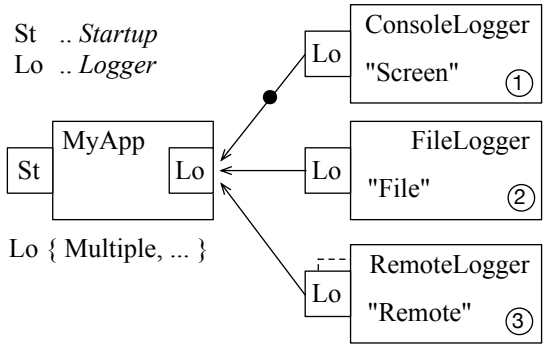


Figure 43. "Logger" sample application for select behavior

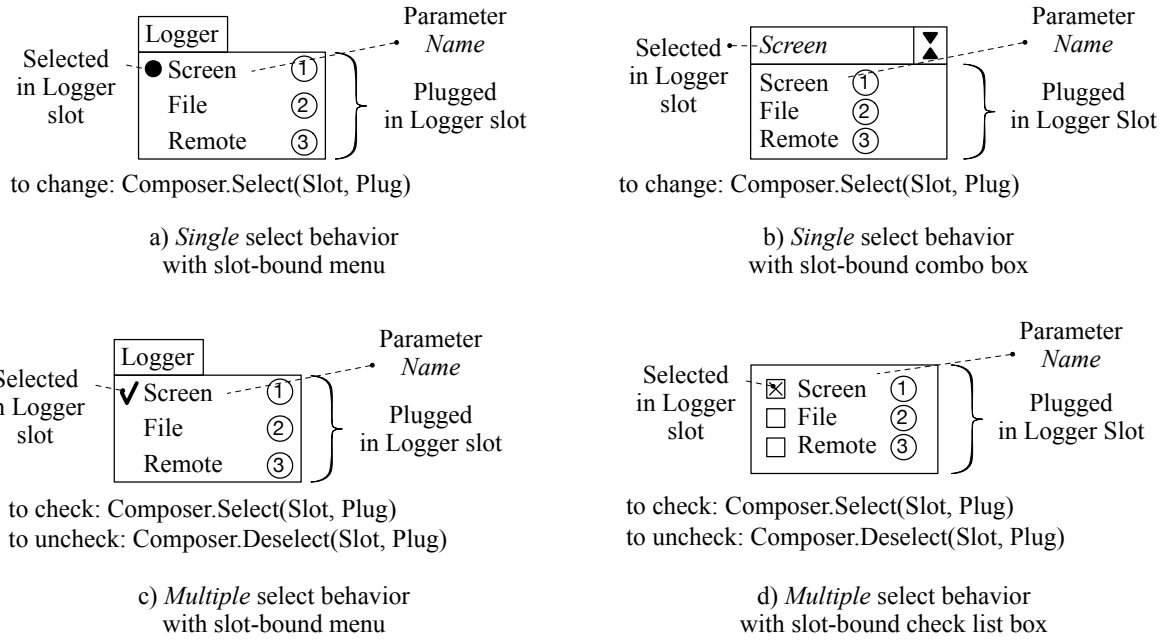


Figure 44. Slot-bound widgets for select behavior

Another widget that can be bound to a slot with single select behavior is the tab widget. In particular, a tab widget can be bound to a Control slot. A control contributor provides the contents for a tab. The control contributors can be loaded lazily, because the tab content is not required, until the tab is actually activated. With lazy loading, the single select behavior, handles registration events on the control slot. When a contributor is registered, the behavior creates a tab, and when a contributor is deregistered, it removes the tab. When a tab is activated, the behavior loads the contributor lazily using the composition core, and selects the contributor. Fig. 45 schematically shows how the tab widget is bound to a control slot.

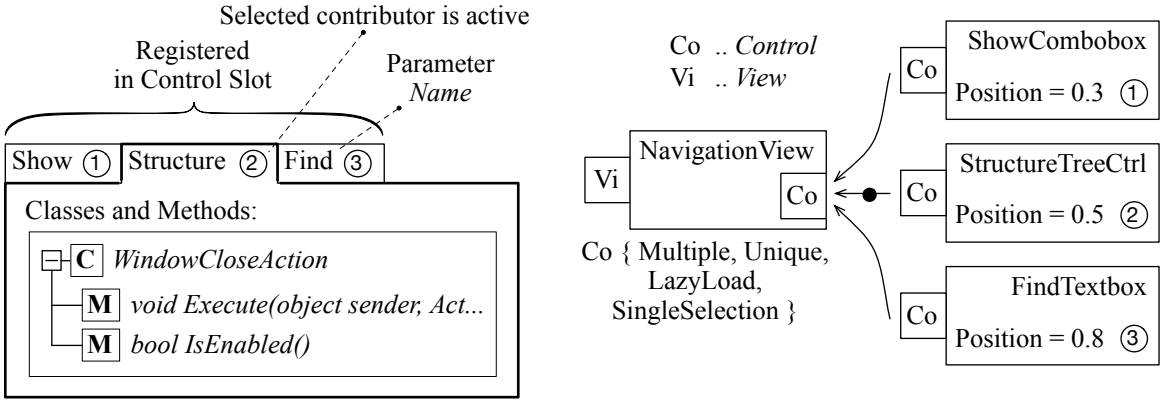


Figure 45. Slot-bound tab widget for single select behavior

5.6 Case Study: Cross-Compiler and IDE

For evaluation, we have used the Plux.NET composition framework in a case study (Jahn 2009b). In this case study we have built a cross-compiler and an integrated development environment (IDE). The cross-compiler should translate Borland Delphi source code into C# source code in an ongoing project with our industry partner BMD Systemhaus GmbH. With a Plux.NET-based design, we wanted to achieve the following goals:

- The cross-compiler should be configurable for arbitrary source and destination languages. Hence, the name of the application is Any-to-Any-Compiler (ATAC).
- The IDE should demonstrate the best practices for dynamic user interface design from Section 5.4 and use the slot-bound widgets from Section 5.5.

5.6.1 Compiler Design

The Atac application is designed around a core *Compiler* extension. The *Compiler* extension uses slots for the frontend, for internal data structures, and for the backend. The Atac application can be used as a console application or within the IDE. Both user interfaces integrate the *Compiler* extension with its *Compiler* plug. Fig. 46 shows the cross-compiler setup for Delphi to C#. The compiler can be extended to a Java to C# compiler with a Java parser plug-in. Or it can be extended to a Delphi to Java compiler with a Java code generator plug-in.

5.6.2 IDE Design

The main window of the Atac IDE is divided into sections (see Fig. 47): The *Control* section on the top contains the user interface to control the compiler. The *Source*, *Data*, and *Output* sections in the center contain views which visualize the contents of the source buffer, the data structures, and the output buffer. The *Addon* section on the bottom contains additional views, for example, an error list view.

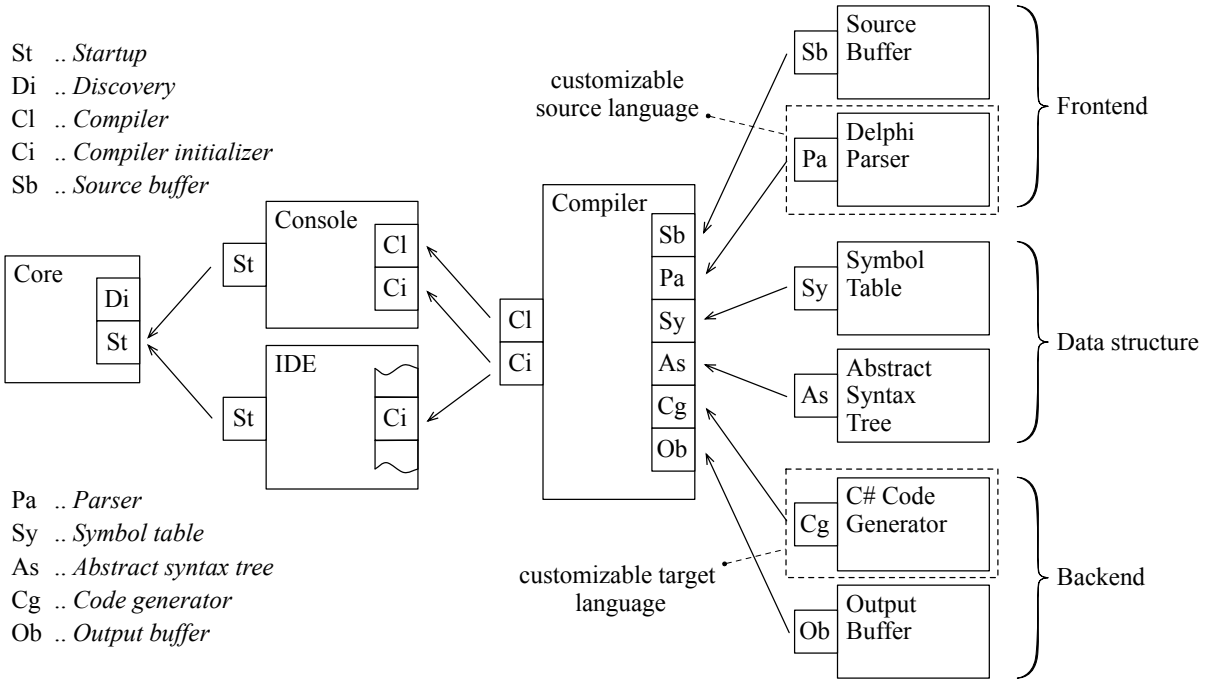
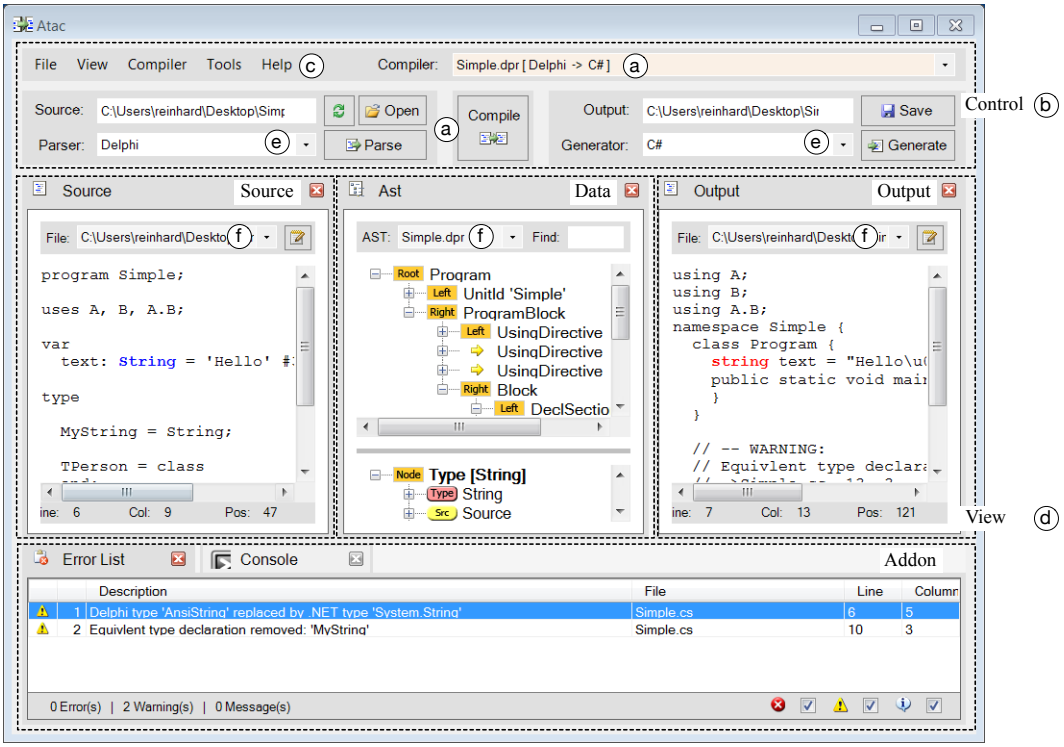


Figure 46. Architecture of extensible ATAC cross-compiler

The design of the Atac IDE uses Plux.NET slots as described in Section 5.1.2. We applied the best practices from Section 5.4 and applied slot-bound widgets from Section 5.5. The key slots and extensions of the Atac IDE are (compare Fig. 47, letters a-f reference markings in the figures):

- The *Compiler slot* and the *Compiler initializer* slot (a) separate the compiler core from the user interface. This is a variant of the *DataSource* slot with unique contributor extensions. The IDE main window initializes the compiler extension. The contributors in the *Control* and *View* slots access the compiler like a data source. The IDE can control multiple compilers in the Compiler slot. A slot-bound combo box with single select behavior in the menu strip extension switches between compilers.
- The *Control* slot (b) makes the *Control* section on the top of the main window customizable. The IDE layouts plug contributors stacked in the order specified by the *Position* parameter.
- The *Action* slot (c) makes the menu strip extension customizable. The menu strip extension uses a slot-bound menu strip with multiple plug behavior to integrate compiler actions in the *Compiler* menu, and buffer actions in the *File* menu. All actions are loaded lazily.



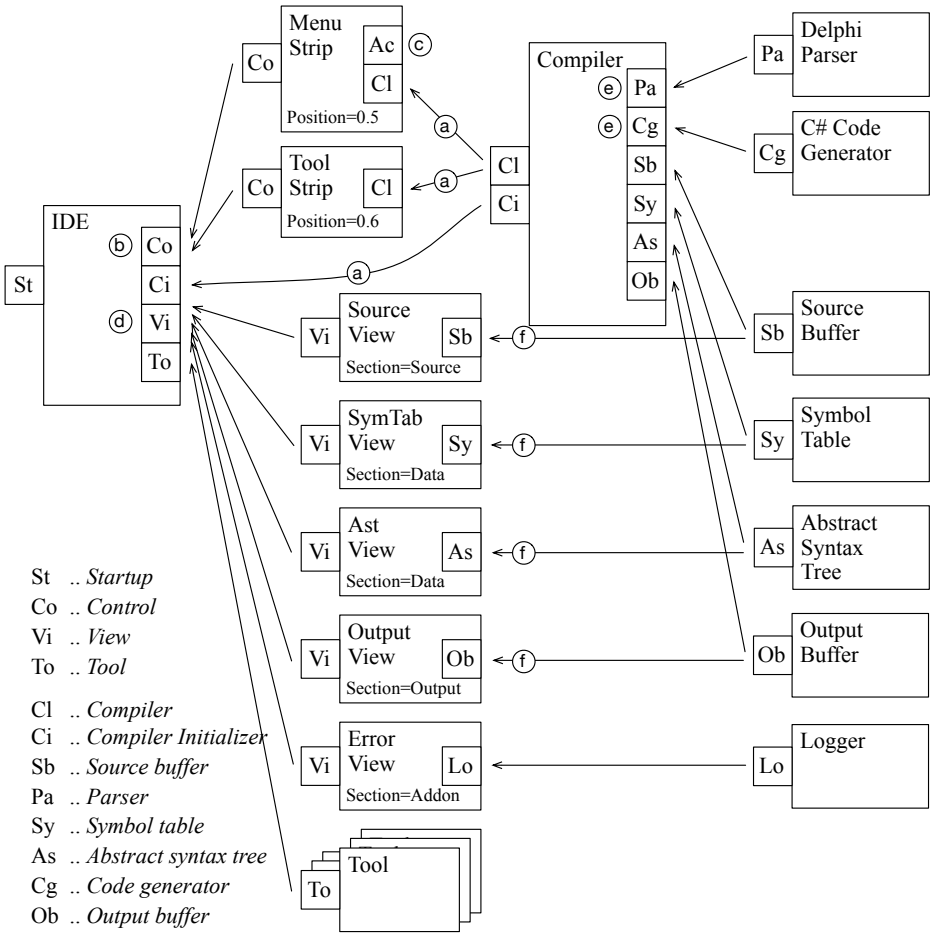


Figure 47. ATAC Integrated Development Environment

- The *View* slot (d) makes the *Source*, *Data*, *Output*, and *Addon* section in the main window customizable. This view slot is an extended variant of the best practice View slot. The IDE layouts plug contributors into four sections as specified by the *Section* parameter. Each section can have multiple view contributors. The IDE uses a variant of the slot-bound tab widget to display multiple views. The slot-bound menu strip in the menu extension binds the *View* menu to the View slot using a multiple plug behavior. And it binds the *Window* menu to the View slot using a multiple select behavior.
- The *ToolStrip* extension uses two slot-bound combo boxes (e) with a single select behavior to switch parsers and code generators.
- The *SourceView*, *SymTabView*, *AstView*, and *OutputView* use slot-bound combo boxes (f) with a single select behavior to switch between *SourceBuffers*, *SymbolTables*, *AbstractSyntaxTrees*, and *OutputBuffers* respectively.
- The *Tool* slot is a variant of the *View* slot and integrates contributors that come in their own window outside of the IDE. Examples for such tools are the Plux.NET Visualizer and the Plux.NET Console.

More information on the design and implementation can be found in the master thesis of Markus Jahn (Jahn 2009b).

Chapter 6: Summary

This chapter summarizes the main contributions of this thesis and recapitulates how those contributions address the problem statement from Chapter 1. Finally, the thesis is concluded with an outlook on future work.

6.1 Contributions

In this thesis, we presented Plux.NET - a novel composition model and prototypical composition infrastructure. The benefits of Plux.NET are dynamic addition and removal of components without programming or configuration. The key characteristics of Plux.NET which enable dynamic change are:

1. A component model in which requirements and provisions between components are specified declaratively using the component's metadata.
2. A discovery core which supports automatic discovery of components using exchangeable discovery mechanisms.
3. A composition core which uses the metadata to compose an application by matching requirements and provisions, and which stores connections between components in the composition model.
4. An event-based programming model, which gives host components a uniform mechanism to integrate contributor components at startup as well as at run time when an application dynamically changes.
5. Best practice guidelines for the design of user interfaces that support dynamic change by reacting to notifications from the composition core and by using component connections stored in the composition model.
6. Slot-bound widgets which automatically update their content and state when the application is reconfigured, because they are bound to slots in the composition model.

6.2 Conclusions

The research context in Section 1.1 discussed why support for dynamic reconfiguration is important. The problem statement in Section 1.2 stated four problems in existing plug-in systems which make it hard to build reconfigurable applications. The contributions of this thesis address these problems:

- *Problem 1: Lack of granularity*.

  The problem has been solved, because the Plux.NET composition core supports fine-grained composition operations. The composition model specifies how to construct extensions with multiple plugs, and how to construct plug-ins containing multiple extensions. The composition core provides operations for each granularity level: it can add or remove a plug, an extension, or a plug-in with a single operation. Additionally, the composition core allows users to control the affected scope of an operation: it can add or remove a contributor to a specific slot, to all slots of a host extension or to all compatible slots in an application.

- *Problem 2: Plug-in integration requires programmatic effort*.

  The problem has been solved, because the Plux.NET composition model replaces programmatic integration in the host component through declarative specification of requirements and provisions. The composition algorithms have been implemented in the Plux.NET composition core, whereas the plug-ins use an event-based programming model. When a host plug-in uses a Plux.NET slot to integrate contributor plug-ins, it must react to composer events, instead of programmatically looking for contributors.

- *Problem 3: Dynamic change support is optional*.
  Problem 4: Non-uniform programming model for startup and dynamic change.

  Both problems have been solved, because the event-based programming model of Plux.NET makes dynamic integration the prevailing mechanism, and at the same time it gives host components a uniform mechanism to integrate other components at start-up as well as at run time when an application changes. When a host plug-in uses a Plux.NET slot to integrate contributor plug-ins, the composer notifies the host when a contributor is added or removed. The composer notifications are treated uniformly, be it at startup, or when the application is reconfigured at run time. A host component does not consider the difference.

6.3 Future Research

Applications designed for the Plux.NET composition infrastructure are extensible and customizable. As we have argued in Chapter 1, that opens interesting usage scenarios. However, the openness and flexibility also means new challenges, which should be addressed by future research:

1. The flexibility of Plux.NET applications makes testing more difficult. Unit tests of single components become less significant, because the functionality of a component can only be observed when composed with other components. Thus integration tests

become more important, but also more challenging, because Plux.NET applications change dynamically. The dynamic properties of a host extension, i.e. how it behaves when contributors are plugged or unplugged, must also be tested. Another testing problem arises from the countless different configurations which must be tested, if each user configures an application differently. A procedure which finds the relevant subset of test cases, is an open research issue.

2. In an open plug-in system, where third-parties contribute their extensions, versioning is an important issue. New versions of a core application should be compatible with plug-ins written for an older version of the core application. Vice-versa, plug-ins written for a new version of the core application, should be compatible with older versions of the core application. How to make a Plux.NET slot versionable, for example by using versioned contracts and adapters, is an open research problem.
3. An open plug-in system allows third parties to contribute functionality. The creator of a host application might want to restrict what a plug-in can do and what it cannot do. Existing plug-in systems cannot restrict composition aspects, as would be required, for example, to control which contributor can contribute to which parts of the system. We have designed a prototype security extension for the Plux.NET framework which allows developers to restrict who is allowed to contribute to a slot, or to restrict the permissions of plug-ins, which connect to a specific slot. We plan to continue this work.
4. In the enterprise domain, customers expect high availability of systems. Integrating plug-ins that have been contributed by unknown third parties can represent an unpredictable risk for the stability of the system. Existing plug-in systems do not offer a solution for this problem. We have designed a prototype isolation extension for the Plux.NET framework which allows developers to protect a host against crashes of buggy or malicious contributors by taking specific precautions for isolating the contributor from the rest of the application.

6.4 Current State

The Plux.NET composition framework prototype for rich client applications is publicly available (http://ase.jku.at/plux). Several ongoing and completed student projects have used Plux.NET. In a pilot project with our industry partner BMD Systemhaus GmbH, we currently develop a prototype for a customizable next generation of BMD business software.

Bibliography

(Basili 1993) Basili, V.R.: The Experimental Paradigm in Software Engineering. Springer-Verlag, #706. Lecture Notes in Computer Science, 1993.

(Beck and Gamma 2003) Beck, K., and Gamma, E.: Contributing to Eclipse. Addison-Wesley, 2003.

(Boudreau et al. 2007) Boudreau, T., Tulach, J., Wielenga, G.: Rich Client Programming, Plugging into the NetBeans Platform, 2007.

(Chatley et al. 2004) Chatley, R., Eisenbach, S., and Magee, J.: Magic Beans: a Platform for Deploying Plugin Components. 2nd International Working Conference on Component Deployment (CD 2004), Edinburgh, Scotland, UK, May 20-21, 2004.

(Councill and Heineman 2001) Heinemann, G. and Councill, W.: Definition of a Software Component and Its Elements. In: Component-Based Software Engineering. Addison-Wesley, Boston 2001.

(Dhungana 2006) Dhungana, D.: CAP.NET - Client Application Platform in .NET. Master Thesis, Johannes Kepler University, Linz, Austria, 2006.

(Eclipse 2003) Eclipse Platform Technical Overview. Object Technology International, Inc., http://www.eclipse.org, February 2003.

(ECMA 2006) ECMA International Standard ECMA-335. Common Language Infrastructure (CLI), 4th Edition, June 2006.

(Eder 2008) Eder, M.: Content-Watcher, ein Werkzeug zur Überwachung von Web-Inhalten. Master Thesis, Johannes Kepler University, Linz, Austria, 2008.

(Floch et al. 2006) Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjørven, E.: Using Architecture Models fpp. 62-70, 2006.

(Fowler 2004) Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html, 2004.

(Hall and Cervantes 2004) Hall, R. S, and Cervantes H: An OSGi Implementation and Experience Report. Consumer Communications and Networking Conference (CCNC), Las Vegas, USA, January 5-8, 2004.

(Hallenstein et al. 2006) Hallsteinsen, S., Stav, E., Solberg, A., and Floch, J.: Using Product Line Techniques to Build Adaptive Systems. Proceedings of the 10th iference, Washington, DC, August 21-24, 2006, pp. 141-150.

(Jahn 2009a) Jahn, M.: Entwurf und Implementierung eines Cross-Compilers von Delphi nach C#. Master Thesis, Johannes Kepler University, Linz, Austria, 2009.

(Jahn 2009b) Jahn, M., Wolfinger, R., and Mössenböck, H.: Extending Web Applications with Client and Server Plug-ins. Software Engineering 2010 - the Conference on Software Engineering, SE 2010, Paderborn, Germany, February 22-26, 2010 (to be published).

(Meyer and Mingins 1999) Meyer, B. and Mingins, C. (eds.): Special Issue on Component-Based Development. IEEE Computer, 82 (7), July 1999.

(Noyes 2006) Noyes, Brian: Data Binding with Windows Forms 2.0: Programming Smart Client Data Applications with .NET 2.0. Addison-Wesley, 2006.

(OSGi 2006) OSGi Service Platform, Release 4. The Open Services Gateway Initiative, http://www.osgi.org, July 2006.

(Parnas 1972) Parnas, D.L.: On the Design and Development of Program Families. IEEE Transactions on Software Engineering, March 1976, pp 1-9.

(Pichler 2009) Pichler, R.: Metrix - A Measuring Tool for Run-time Figures in Plug-in-based .NET Applications. Bachelor Thesis, Johannes Kepler University, Linz, Austria, 2009.

(Pico 2009) PicoContainer Committers: PicoContainer 2.8.1 Documentation. http://www.picocontainer.org., 2009.

(Rabiser 2009) Rabiser, R., Wolfinger, R., Grünbacher, P.: Three-level Customization of Software Products Using a Product Line Approach. 42nd Hawaii International Conference on System Sciences, HICSS-42, Big Island, Hawaii, USA, January, 5-8, 2009.

(Reiter 2007) Reiter, S., Wolfinger, R.: Erfahrungen bei der Portierung von Delphi Legacy Code nach .NET. Nachwuchs-Workshop, SE 2007 - the Conference on Software Engineering, Hamburg, Germany, March 27-30, 2007.

(Sells and Weinhardt 2006) Sells, C., Weinhardt, M.: Windows Forms 2.0 Programming. Addison-Wesley, 2006.

(Sun 1996) Sun Microsystems: JavaBeans, Version 1.0. http://java.sun.com/beans. December, 1996.

(Sun 2006) Sun Microsystems: Java Platform, Standard Edition 6, API Specification. http://java.sun.com/javase/6/docs, 2006.

(Szyperski 2002) Szyperski, C.: Component Software, Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, 2002.

(Weinreich and Sametinger 2001) Weinreich, R., Sametinger, J.: Component Models and Component Services: Concepts and Principles. In: Component-based Software Engineering, 2001.

(Wolfinger 2006) Wolfinger, R., Dhungana, D., Prähofer, H., Mössenböck, H.: A Component Plug-in Architecture for the .NET Platform. Modular Programming Languages, Lightfoot, David; Szyperski, Clemens (Eds.), Lecture Notes in Computer Science , Vol. 4228, Proceedings of 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006.

(Wolfinger 2007) Wolfinger, R., Prähofer, H.: Integration Models in a .NET Plug-in Framework. SE 2007 - the Conference on Software Engineering, Hamburg, Germany, March 27-30, 2007.

(Wolfinger 2008a) Wolfinger, R., Reiter, S., Dhungana, D., Grünbacher, P., and Prähofer, H.: Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. 7th IEEE International Conference on Composition-Based Software Systems, ICCBSS 2008, Madrid, Spain, February, 25-29, 2008.

(Wolfinger 2008b) Wolfinger, R.: Plug-in Architecture and Design Guidelines for Customizable Enterprise Applications, OOPSLA 2008 Doctoral Symposium, OOPSLA 2008, Nashville, Tennessee, October, 19-23, 2008.

List of Figures

List of Tables

Curriculum Vitae

Name:	Reinhard Wolfinger
Date of birth:	January 22, 1972
Place of birth:	Linz, Austria
Nationality:	Austria
Marital status:	Married, 2 children
Contact:	reinhard.wolfinger@jku.at | reinhard.wolfinger@gmail.com

Education

2006-2010	Doctorate Program in Social Sciences, Economics and Business, Johannes Kepler University, Linz
2003-2005 1991-1992	Diploma Study Business Informatics, Johannes Kepler University, Linz Graduated with distinction Graduation degree: Magister rer.soc.oec. Master thesis: Spyder.NET - Automated recording of use cases for testing of .NET components
1986-1991	Higher technical school for Communications Engineering, Leonding Graduated with distinction
1982-1986	Grammar school, Traun
1978-1982	Primary school, Haid

Professional Career

2006-	Research Assistant, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University, Linz
2005	Software Developer, BMD Systemhaus GmbH, Steyr
2004-2005	Self-employed Software Developer
2001-2003	Software Development Lead, AgrarData GesmbH, Linz
1991-2001	Self-employed Software Developer