

Adding Genericity to a Plug-in Framework

Reinhard Wolfinger¹, Markus Löberbauer², Markus Jahn², Hanspeter Mössenböck^{1,2}

Christian Doppler Laboratory for Automated Software Engineering²

Institute for System Software¹

Johannes Kepler University, Linz, Austria

reinhard.wolfinger | markus.loeberbauer | markus.jahn | hanspeter.moessenboeck@jku.at

Abstract

Plug-in components are a means for making feature-rich applications customizable. Combined with plug-and-play composition, end users can assemble customized applications without programming. If plug-and-play composition is also dynamic, applications can be reconfigured on the fly to load only components the user needs for his current work. We have created Plux.NET, a plug-in framework that supports dynamic plug-and-play composition. The basis for plug-and-play in Plux is the *composer* which replaces programmatic composition by automatic composition. Components just specify their requirements and provisions using metadata. The composer then assembles the components based on that metadata by matching requirements and provisions. When the composer needs to reuse general-purpose components in different parts of an application, the component model requires genericity. The composer depends on metadata that specify which components should be connected and for general-purpose components those metadata need to be different on each reuse. We present an approach for generic plug-ins with component *templates* and an implementation for Plux. The general-purpose components become templates and the templates get parameterized when they are composed.

Categories and Subject Descriptors D.2.11 [Software Engineering]: Software Architectures - Patterns.

General Terms: Design.

Keywords: Generic plug-ins, Component templates, Plug-and-play composition, Run-time adaptation; Plug-in architectures; Composition, Generic programming

1. Introduction

With plug-and-play composition, end users can assemble applications without programming. This can be used to customize a feature-rich application to the needs of individual users. Combined with dynamic composition, an application can be reconfigured on the fly to load only components the user needs for his current work. This keeps an application small and aligned with the working situation at hand.

Plux is a novel plug-in framework with a composition model and an infrastructure for plug-and-play composition [1]. The composition model specifies requirements and provisions among components declaratively using the component's metadata. The infrastructure contains a composer

which assembles an application by matching requirements and provisions.

In many applications, the composer needs to reuse general-purpose components in different parts of the application. For example, a general-purpose data grid might use different contributor components for its grid layout or as a data source on each reuse (cf. Figure 1). In programmatically composed applications, the programmer creates and connects the objects. In plug-and-play composed applications, however, the programmer does not have control over the composition process. Instead, the composer in the composition infrastructure assembles the components. Thereby the composer depends on metadata that specify which components need to be connected. If we have general-purpose components that require different metadata on each reuse, the composition model requires genericity. The components become templates and the templates must be parameterized when they are composed. In this paper, we present an approach for generic plug-ins with component templates and an implementation for the Plux plug-in framework.

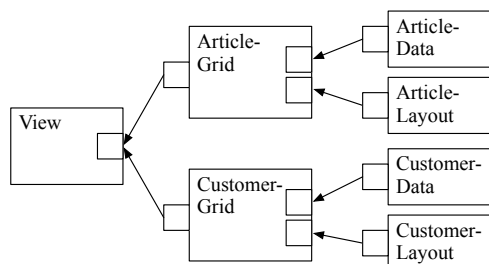


Figure 1. Data grid reused for articles and customers.

The paper is organized as follows: Section 2 describes the Plux framework. It highlights the metadata model and the architecture of the composition infrastructure. Section 3 gives a motivating example, discusses the problems that arise with this example in existing plug-in systems, and outlines the requirements for a solution. Section 4 describes our generic plug-in approach, the integration of generic plug-ins into Plux, the notation for extension templates, and the extension generation using metadata from a configuration file. Section 5 describes a case study where we applied generic plug-ins in an enterprise application of our industrial partner. Section 6 describes general approaches to genericity and how existing plug-in systems handle genericity. Section 7 finishes with a conclusion and an outlook to further work.

2. The Plux.NET Framework

The goal of Plux is to create extensible and customizable applications that can be reconfigured without a restart. To enable such applications, Plux defines a composition model and an infrastructure for dynamic composition. Dynamic composition allows developers to build applications where users load and integrate only components they need for their current work. Dynamic composition also means that an application can be reconfigured on the fly by dynamically swapping sets of components without programming or configuration.

When compared with other plug-in systems [2], such as OSGi [3], Eclipse [4], or NetBeans [5], the unique characteristics of Plux are the *composer*, the *event-based programming model*, the *composition state*, and the exchangeable *plug-in discovery mechanism*. The composer replaces programmatic composition where the host component queries a service registry and creates and integrates its contributors itself. In Plux, the components just declare their requirements and provisions using metadata. The composer then uses those metadata to match requirements and provisions and automatically integrates matching components. During composition, the host components react to notification events sent by the composer. The composition infrastructure stores the composition state, i.e., it stores which host components use which contributor components. Unlike in other plug-in systems, the plug-in discovery mechanism is not an integral part of the Plux infrastructure, but is a plug-in itself, thus making the mechanism replaceable. The discovery plug-in is responsible for detecting plug-ins and extracting component metadata. The following subsections cover those characteristics in more detail.

2.1 Meta elements

The Plux composition model (CM) uses the metaphor of extensions with slots and plugs. An extension is a functional component which provides services to other extensions and uses services provided by other extensions. As Figure 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 plugs and slots. An extension which opens a slot is called a host extension, whereas an extension filling a slot is called a contributor extension.

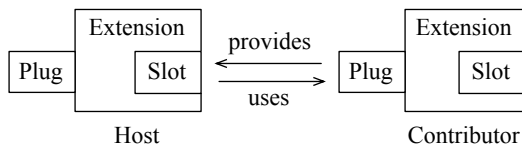


Figure 2. Extensions, slots and plugs.

The host and the contributor communicate via a defined protocol to accomplish a particular task. Every slot has a slot definition which specifies an interface that is required for the collaboration. The host relies on this interface and the contributor has to provide an implementation for it. A slot definition can specify additional parameters for which

the contributor has to provide values (cf. Section 2.3). A slot definition is referenced by its unique name.

The CM uses meta elements to describe extensions and their relationships. In a Plux application, there is an *Extension* meta element for every pluggable .NET object (cf. Figure 3). The *Object* property of the extension references the .NET object in a one-to-one relationship. If a host wants to use the services of contributor extensions, it requires a *Slot* meta element. If a contributor wants to provide a service to hosts, it requires a *Plug* meta element. Both, the slot and the plug are identified by their name. This name references the corresponding slot definition. A plug matches a slot, if their names match. A matching plug can be plugged into the slot, if the slot definition is available, and if the plug qualifies for the slot definition. A plug qualifies, if it provides an implementation for the required interface as well as parameter values for the required parameters.

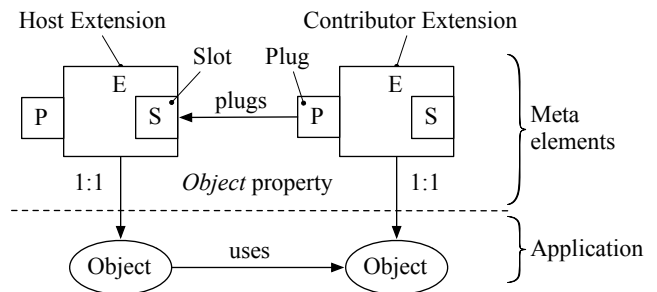


Figure 3. Relationships between meta elements and application objects.

2.2 Composition relationships

Composition means the mediating process which matches required and provided services, or in other words to compose an application by plugging plugs into slots. After a contributor is plugged into a host, the host is notified that the contributor is ready to be used (cf. Figure 4a).

The CM activates contributors lazily. When a contributor is plugged into its host, the contributor's .NET object is not yet instantiated. Only when the host accesses the *Object* property of the contributor, the actual object is created. The contributor is now *activated* and the host can call methods from the contributor's interface (cf. Figure 4b).

A slot can have multiple contributors plugged. If a host wants to use only a subset of them, or if it wants to switch between contributors, it can set a selection on one or several contributors. The slot meta element gives the host access to the *selected* contributors (cf. Figure 4c).

Contributors can be *shared* or *unique*. A unique contributor connects to just a single slot, whereas a shared contributor can be plugged into several slots. For every contributor class there is just a single shared instance in the whole application. Slots can declare whether they want the composer to connect them with this single shared contributor or a with a new unique contributor.

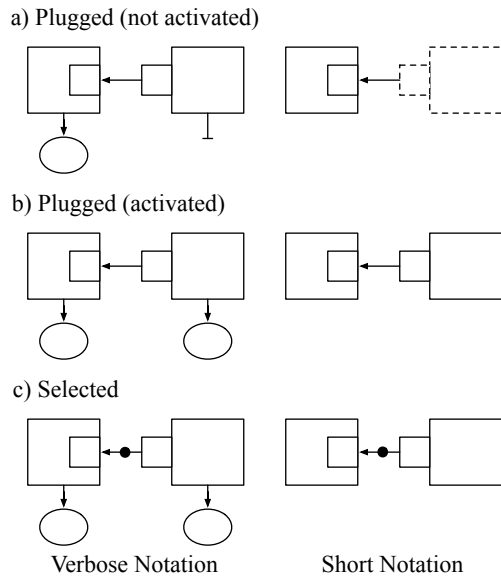


Figure 4. Relationships in the Plux composition model.

2.3 Custom attributes

The mechanism to declare metadata in Plux is customizable. The default mechanism which is included in the framework, declares meta elements with custom .NET attributes. Custom attributes are pieces of meta-information that can be attached to language constructs, such as classes, interfaces, methods, or fields in the source code of a .NET application. At run time, the attributes can be retrieved using reflection [6]. As custom attributes declare metadata directly in the source code, they allow us to avoid separate files, like the XML files used in Eclipse [4].

Let us look at an example. Assume that a host wants to print log messages with time stamps. The logging should be implemented as a contributor that plugs into the host. The contributor should provide the desired format for the time stamp as a parameter to the host. First, we have to define the slot into which the logger can plug.

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

Listing 1. Definition for the Logger slot.

The slot definition in Listing 1 is a C# interface tagged with a `[SlotDefinition]` attribute specifying the name of the slot ("Logger"). The `[ParamDefinition]` attribute specifies a parameter `TimeFormat` of type `string`. The contributor will provide a time format and the host will use it to include the formatted time stamp in the log message. Next, we are going to write a contributor that fits into a `Logger` slot.

The contributor in Listing 2 is a C# class tagged with an `[Extension]` attribute specifying the name of the contributor. If the name is omitted in the attribute, the contributor adopts the class name. The class implements the interface

`ILogger` of the corresponding slot definition. The `[Plug]` attribute defines a plug that fits into the `Logger` slot. The `[Param]` attribute sets the value "hh:mm:ss" for the parameter `TimeFormat`.

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

Listing 2. Console logger contributor for the `Logger` slot.

Finally, we implement the host. The host is an extension which plugs into the `Application` slot of the Plux core. The host in Listing 3 has a slot `Logger`. This is declared with a `[Slot]` attribute. The slot is configured for multiple and unique contributors, because *multiple* and *unique* are the default settings for a slot.

In the constructor, `MyApp` gets a reference to the associated extension meta object and retrieves the slot named "Logger". Then it starts a separate thread, where the actual work is done.

```
[Extension]
[Plug("Application")]
[Slot("Logger")]
public class MyApp : IApplication {
    Slot loggerSlot;
    public void MyApp(Extension e) {
        loggerSlot = e.Slots["Logger"];
        new Thread(Exec).Start();
    }
    void Exec() {
        ILogger logger;
        string format;
        while(true) {
            string msg;
            DoWork(out msg);
            foreach(Plug p in loggerSlot.PluggedPlugs) {
                logger = (ILogger) p.Extension.Object;
                format = (string) p.Params["TimeFormat"];
                logger.Print(DateTime.Now.ToString(format)
                    + ":" + msg);
            }
            Thread.Sleep(2000);
        }
    }
    void DoWork(out string msg) {
        /* not shown */
    }
}
```

Listing 3. Application host with the `Logger` slot.

In the `Exec` method, the host does its work and then uses the plugged loggers to print a message. The contributors can be accessed via the `PluggedPlugs` collection of the logger slot. For each logger, the host accesses the .NET object through the property `Object` and retrieves the logger's time format from the parameter `TimeFormat`. Then it formats the time stamp and prints the log message. The thread repeats that operation in a two second interval. This host implementa-

tion supports dynamic reconfiguration. If at run time, loggers are dynamically added or removed, the host reflects the configuration change, because the composition model updates the `PluggedPlugs` collection.

This completes the example. We compile the slot definition with the interface `ILogger` in a separate DLL file (the so-called contract), because both, the host and the contributor, compile against the interface `ILogger`. If we compile the classes `ConsoleLogger` and `MyApp` into plug-in DLL files and drop them into the plug-in repository of Plux everything will fall into place. The Plux infrastructure will discover the extension `MyApp` and plug it into the `Application` slot of the Plux core. It will also discover the extension `ConsoleLogger` and plug it into the `Logger` slot of `MyApp` (cf. Figure 5).

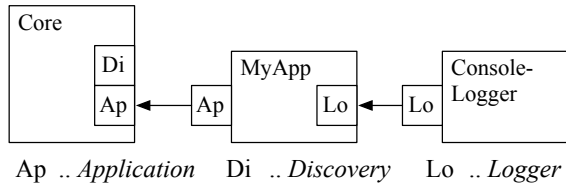


Figure 5. Composed application with host and logger contributor.

2.4 Composition infrastructure

The composition infrastructure (CI) allows the execution of applications built from contracts and plug-ins that conform to the composition model. Figure 6 shows the subsystems of the CI and the way how composition works. In a nutshell, the CI discovers extensions in a plug-in repository and composes an application from them by connecting matching slots and plugs.

The plug-in repository is typically a folder in the file system containing contract DLL files (i.e., slot definitions) and plug-in DLL files (i.e., extensions).

The discovery core ensures that at any time the type store contains the metadata representation of the plug-in repository. When the discovery core detects an addition to the repository, it extracts the metadata from the DLL file and adds it to the type store. Vice versa, when it detects a removal from the repository, it removes the corresponding metadata from the type store.

The type store maintains type metadata for contracts and extensions that are available for composition. It acts as an observable object notifying the composition core about changes. Thus, whenever new types become available, or when types are no longer available the composition core can take appropriate measures.

The composition core (short: composer) assembles the application by matching slots and plugs. For this purpose it observes the type store for changes. If a contributor becomes available in the type store, the composer queries the instance store for matching slots. If it finds a matching slot and a plug of the contributor qualifies, the composer plugs the contributor. To plug a contributor means to instantiate it, add it to the instance store, and add a *plugged* relationship between the host and the contributor to the instance store. After that, the composer opens the slots of the con-

tributor. Thus, the contributor becomes itself a host and the composer fills its slots. Vice versa, if a contributor is removed from the type store, the composer queries the instance store for relationships containing the contributor's plugs. If it finds such relationships, it unplugs the contributor. To unplug a contributor means to close its slots, to remove the *plugged* relationship from the instance store, to remove the contributor from the instance store, and to release it. Closing the contributor's slots causes the decomposition to be propagated, i.e., all contributors are then unplugged from those slots as well.

In other words, the instance store maintains the current composition state of an application, i.e., the instance metadata for extensions and their relationships.

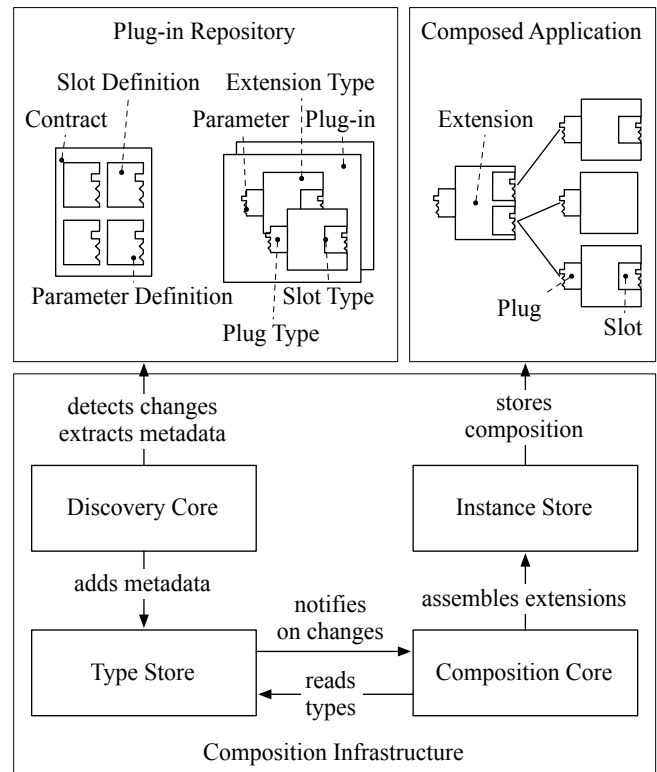


Figure 6. Subsystems and responsibilities of the Plux composition infrastructure.

2.5 Extensible discovery

Discovery comprises that part of the Plux CI which is responsible for detecting plug-ins and extracting metadata from them. Unlike in other plug-in systems, the Plux discovery mechanism is not an integral part of the CI, but is a plug-in itself. The CI's discovery core merely contains the infrastructure necessary for integrating external discoverer extensions (short: discoverer) that again have slots for detector and analyzer extensions (short: detector, analyzer). Discoverers plug into the *Discovery* slot of the Plux core.

When the discovery core integrates a discoverer, it provides a type builder, which allows the discoverer to build meta objects compatible with the type store. After a discoverer has detected a plug-in and has provided meta ob-

jects for it, it adds these meta objects to the type store.

The discovery core is designed for dynamic discovery. Thus, while an application is running, the discoverer can monitor a repository in a separate thread, and when it detects changes, it calls back into the discovery core which in turn updates the type store.

For bootstrapping, Plux includes a default discoverer plug-in which includes a startup detector (cf. Figure 7). When this discoverer is integrated into the discovery core, its startup detector inspects a set of folders and files for plug-ins. Those files and folders can be specified as command-line arguments when Plux is launched.

The default discoverer plug-in contains also an attribute analyzer which extracts type metadata from custom attributes (cf. Section 2.3) in plug-in DLL files.

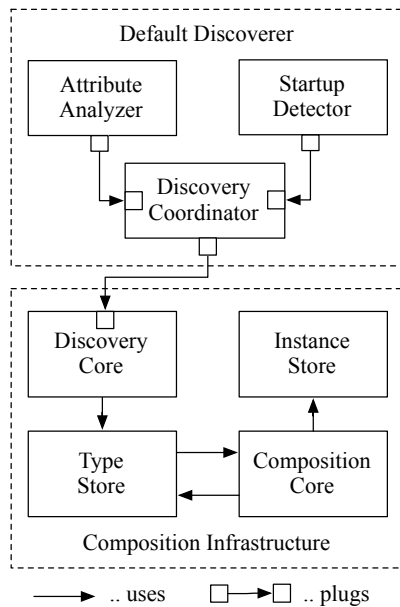


Figure 7. Integration of the default discoverer into the discovery core.

Inside the default discoverer, the discovery coordinator coordinates detectors and analyzers. The detectors inspect repositories and detect plug-ins. The analyzers extract metadata from plug-ins. The coordinator is also extensible. Thus, to customize discovery, we can either contribute a complete discoverer to the discovery core, or we can contribute a detector or an analyzer to the coordinator. Keeping the detector and the analyzer as separate extensions allows us to replace them individually. For example, we could replace the detector with one that retrieves plug-ins over the network instead of from a file system folder, or we could replace the analyzer with one that gets the metadata of extensions from XML files instead of from .NET attributes.

There can be several detectors and analyzers plugged into the discovery coordinator at the same time, each of them responsible for detecting plug-ins from different sources and for analyzing them according to their structure. For example, in addition to the startup detector (that retrieves plug-ins from the files and folders specified in the

command-line arguments) there is also a repository detector that continuously monitors a special folder (the plug-in repository) for plug-ins. The repository detector is specified in the list of command line arguments and is therefore detected by the startup detector. It is then analyzed and plugged into the discovery coordinator. From now on both the startup detector and the repository detector will be active, each of them trying to retrieve plug-ins from their particular sources.

2.6 More features

Other features of the infrastructure that cannot be discussed here are the management of *composition rights* (e.g., which extensions are allowed to open a certain slot, and which extensions are allowed to fill it), *slot behaviors* that allow developers to specify the way how slots behave during composition (e.g., one can limit the capacity of a slot to n contributors, or one can automatically remove a contributor from a slot when a new contributor is plugged in there), as well as a *scripting API* that allows experienced users to override some of the operations of the composition core. For a more extensive description of the features see [1][7][8].

3. Motivating Example

In Plux, the metadata of the slots and plugs define which extensions can be plugged together by the composition core. As shown in Section 2.3, the default way to specify this metadata is to use .NET attributes attached to language constructs. For general-purpose extensions, which we want to reuse in different parts of the application, the problem is that we need different metadata on each reuse.

Customers ✕

#	NAME	PHONE	STREET	CITY
1	ACME Inc.	(216) 272-0003	40 West Orange Stre	Chog ▲
2	IBM Corp.	(800) 426-9900	1 New Orchard Road	Armc ☰
3	Microsoft C	(800) 426-9900	1 Microsoft Way	Redn ▼

search for: search in: search mode:

Articles ✕

#	CODE	DESCRIPTION	SPECIFIC.	SUPPLIER
1	110-0420	Conveyor Belt	100 x 85	B5000x ▲
2	230-2210	Cardan Joint	90/280 TQY	MB505/A3 ☰
3	700-8310	Petrol Pump	200 oz. / 2hp	ZT200/2b ▼

search for: search in: search mode:

Figure 8. User interface for customer and article view.

Let us assume that we want to create an enterprise resource planning application with a customer view and an article view as shown in Figure 8. Since we want to be able

to add arbitrary controls to the views, the view extensions need a slot for controls. We provide two contributors for this slot: A grid panel, which displays data records, and a filter panel, which offers search capabilities. Both need a slot for the data source that will be plugged into them.

Listing 4 shows the definitions for the `Control` and the `DataSource` slot. The `Control` slot requires a float parameter `Order`. The view host arranges the controls from top to bottom using the floating point value to determine their order.

```
[SlotDefinition("Control")]
[Param("Order", typeof(float))]
public interface IControl {
    Control Control { get; }
    string Name { get; }
}

[SlotDefinition("DataSource")]
public interface IDataSource {
    string Name { get; }
    object Data { get; }
    event EventHandler Changed;
}
```

Listing 4. Definition of control and data source slot.

The grid panel and the filter panel are general-purpose extensions that can be reused at many places of an application. Here, they should be instantiated twice, once for each view. The *Grid* extension has a *Control* plug for the view host (cf. Figure 10), and it has a *DataSource* slot for the data provider. If we declare the metadata with custom attributes as shown in Listing 5, the composer plugs a separate grid into each view, as the *Control* slots in both views require unique contributors. But then we run into two problems: (1) When the composer tries to fill the *DataSource* slots of the grids, it plugs every data contributors into every grid, because the data contributors' plugs match the slots of all grids. But this is not the intended composition (cf. Figure 9). (2) In a similar way, the composer plugs every control into every view. Thus, one cannot specify that a particular control must be plugged into only one of the views, e.g., that the filter panel should only go into the article view, but not into the customer view.

```
[Extension]
[Plug("Control")]
[Param("Order", 0.5f)]
[Slot("DataSource", Shared=true)]
public class Grid : IControl { ... }
```

Listing 5. Metadata for data grid extension.

Figure 10 shows the composition as intended: The *CustomerData* extension (3) plugs only into controls which contribute to the customer view (1), and the *ArticleData* extension (4) plugs only into controls which contribute to the article view (2).

3.1 Problems without generics

In order to compose the customer view and the article view correctly, we need to selectively connect controls with views as well as data sources with controls. Without generics there are two solutions for this problem: (a) Disabling

the composer and plugging extensions together programmatically. (b) Writing different versions of the *Grid* and *Filter* extensions with different metadata that either match the customer view context or the article view context. Let us see how these approaches work and why they are inadequate.

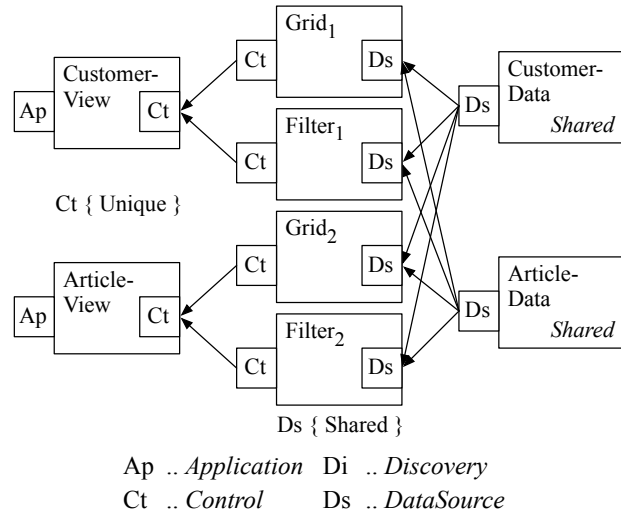


Figure 9. Incorrect composition of data providers.

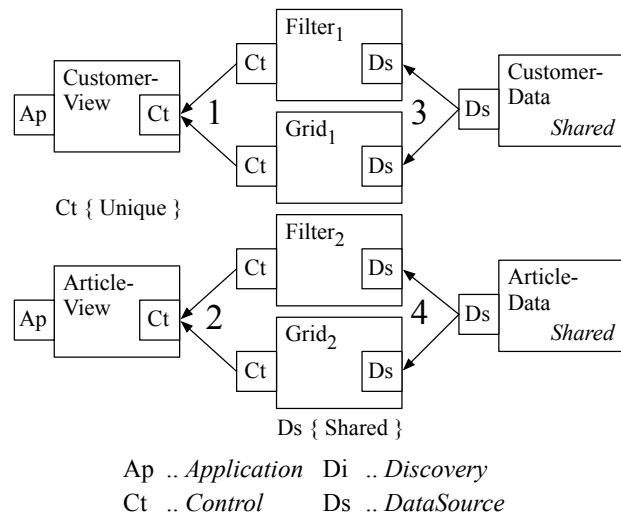


Figure 10. Composed application with customer and article view.

(a) Programmatic composition means to disable the composer for a particular slot, which can be done by setting the slot's `AutoPlug` property to `false`. As a result, the composer will not automatically plug contributors into this slot. Instead, we have to do that manually in the source code. For example, if we want to plug a *Grid* extension into the `Control` slot of `ArticleView` we have to write an event handler for the `opened` event of the `Control` slot, which is raised when this slot is opened (cf. Listing 6). In the event handler, we have to look up the *Grid* extension in the type store, instantiate it, disable the composer for its `DataSource` slot, and plug it into the `Control` slot using a special script-

ing API. Then we have to look up the `ArticleData` extension and plug it into the `DataSource` slot of `Grid`.

```
[Slot("Control", AutoPlug=false,
    OnOpened="Control_Opened")]
public class ArticleView : IApplication {
    Slot controlSlot;
    public ArticleView(Extension e) {
        controlSlot = e.Slots["Control"];
    }
    public void Control_Opened(SlotEventArgs args) {
        TypeStore typeStore = args.Runtime.TypeStore;
        ExtensionType gridType
            = typeStore.ExtensionTypes["Grid"];
        Extension grid = gridType.CreateExtension();
        grid.Slots["DataSource"].AutoPlug = false;
        controlSlot.Plug(grid.Plugs["Control"]);

        ExtensionType dataSourceType
            = typeStore.ExtensionTypes["ArticleData"];
        Extension data
            = dataSourceType.GetSharedExtension();
        grid.Slots["DataSource"].Plug(
            data.Plugs["DataSource"]);
    }
}
```

Listing 6. Programmatic composition in article view host.

Programmatic composition causes three problems: (1) The composition of controls and data sources must be programmed manually, which is exactly the effort we wanted to avoid, when we chose to use the Plux composer in the first place. (2) Programmatic composition of controls renders the view inextensible, because the view cannot integrate controls that were not known at compile time. (3) If all data sources share the same plug name, other hosts also need to compose manually, if they want to access the data source.

Parameter values are an additional problem. For example, the `Control` slot in Listing 4 requires the float parameter `Order`. This floating point value determines the order in which the view host arranges the controls. If we declare the order value with an attribute on the grid, each grid has the same position regardless of the view that it contributes to. But we want each grid to have a different order value.

(b) The second approach is to subclass a new extension on each reuse. In Listing 7 we derive subclasses for the article view and the customer view.

```
[Plug("ArticleControl"),
    Param("Order", 0.5f),
    Slot("ArticleData", Shared=true)]
public class ArticleGrid : Grid { ... }

[Plug("CustomerControl"),
    Param("Order", 0.7f),
    Slot("CustomerData", Shared=true)]
public class CustomerGrid : Grid { ... }
```

Listing 7. Extension reuse through sub-classing.

On each subclass we can declare a unique plug, a different value for the order parameter, and a unique slot for the data source. Although that approach does allow automatic composition, it causes two problems: (1) Subclassing creates many unnecessary types, because we create a new class on

each reuse, when all we need is new metadata. (2) We have to redeclare *all* metadata on the subclass, although we only wanted a new slot and plug name, and different parameter values. For example, the `DataSource` slot in Listing 5 configures the slot for shared contributors (`Shared=true`). This is true for all grids. If we by mistake set a different configuration in a subclass, the extension will work uncorrectly.

3.2 Requirements for generics

From this example and the problems of non-generative approaches, we derive three requirements: (1) It should be possible to generate extensions with metadata that are separated from the implementation class. (2) We want to generate multiple extensions with different metadata from a single class. (3) We would like to distinguish between different kinds of metadata. Metadata that configure the composer such as `Shared=true` or `AutoPlug=false` should not be generic, i.e., they should be exactly the same in all extensions that are generated from a template. On the other hand, metadata that describe slot names, plug names and parameter values should be able to vary in the generated extensions.

4. Generic Plug-ins

Here, we introduce our approach for generic plug-ins that meets the requirements from Section 3.2. A generic plug-in contains one or several *extension templates*. An extension template comprises a class, metadata that configure the composer, and metadata placeholders for slot names, plug names, and parameter values. In contrast to regular extensions, metadata in templates are incomplete because of the placeholders. In order to generate extensions from templates, one has to discover the templates and replace the placeholders with metadata from an external source (e.g., a configuration file or a database).

Figure 11 revisits the example from Section 3 using a template. (a) The template *GridTemplate* has four placeholders: `<Grid>` for the extension name, `<Control>` for the plug name, `<DataSource>` for the slot name, and `<Order>` for a parameter value. (b) The external metadata specify that two extensions should be generated from the template, the *ArticleGrid* and *CustomerGrid* extensions. For the article grid, the *ArticleControl* plug replaces the `<Control>` placeholder, the *ArticleData* slot replaces the `<DataSource>` placeholder, and the float value `0.5` replaces the `<Order>` placeholder. For the customer grid, the placeholders are replaced in the same manner.

Since generic plug-ins only affect the variability of metadata and metadata are provided by the Plux discovery mechanism, generic plug-ins can be integrated into Plux by introducing a custom analyzer for templates (cf. Section 2.5). A template analyzer can be plugged into the discovery coordinator of the default discoverer. It replaces placeholders in the template metadata with external metadata, and as a result, it generates extensions from templates.

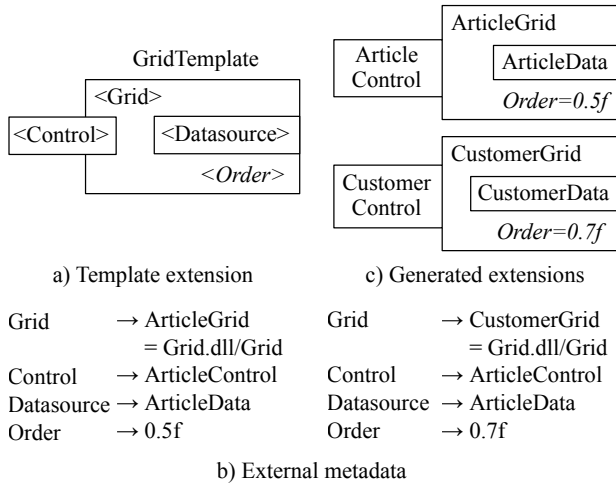


Figure 11. *ArticleGrid* and *CustomerGrid* extensions generated from template *Grid* and external metadata.

4.1 The Template attribute

To declare a template, we attach the *Template* attribute to a class. Listing 8 shows the grid template example. For the plug, the parameter, and the slot we use the corresponding Plux attributes. In order to distinguish placeholders from real meta names, we enclose the placeholder names in chevrons. The names in chevrons serve two purposes: firstly, they are placeholders that are to be replaced with names from the external metadata; secondly, they specify the slot definition on which the named slot or plug is based. For example, the template slot *<DataSource>* is based on the slot definition *DataSource* (cf. Listing 4).

```
[Template("Grid")]
[Plug("<Control>")]
[Param("Order", "<Order>")]
[Slot("<DataSource>", Shared=true)]

public class Grid : IControl { ... }
```

Listing 8. Grid template definition.

```
<Grid> -> ArticleGrid=Grid.dll/Grid
<Control> -> ArticleControl
<Datasource> -> ArticleData
<Order> -> 0.5f
```

Listing 9. Metadata for article grid.

```
[Extension("ArticleGrid")]
[Plug("ArticleControl")]
[Param("Order", 0.5f)]
[Slot("ArticleData",
      SlotDefinition="DataSource", Shared=true)]

public class Grid : IControl { ... }
```

Listing 10. Attributes equivalent to generated article grid.

Listing 10 shows a class and its attributes that are conceptually equivalent to the grid class that results from the template in Listing 8 and the external metadata in Listing 9. In fact, such a class is not generated explicitly, but the class of the template (i.e., *Grid*) is entered directly into the type store, decorated with the external metadata.

For the article grid, `[Template("<Grid>")]` was replaced with `[Extension("ArticleGrid")]`, `[Plug("<Control>")]` with `[Plug("ArticleControl")]`, and `[Slot("<DataSource>")]` with `[Slot("ArticleData")]`. Since the latter slot is actually based on the slot definition *DataSource* we need to set the `SlotDefinition` property of the `[Slot]` attribute in order to map *ArticleData* to *DataSource*. Finally, the parameter placeholder `<Order>` is replaced with the float value `0.5`. For the customer grid, the placeholders are replaced in the same way.

The retrieval of slot meta elements in templates is different than in regular extensions, because the name of a template slot is not known at compile time. Thus, when we have to access this slot from the code of the extension, we use a slot index instead of a slot name. In Listing 11 the slot `<DataSource>` was given the index `0`, and this index is used when the slot is accessed in the *Grid*'s constructor.

```
[Template("Grid")]
[Plug("<Control>")]
[Param("Order", "<Order>")]
[Slot("<DataSource>", Index=0, Shared=true)]

public class Grid : IControl {
    Slot dataSourceSlot;
    public void Grid(Extension e) {
        dataSourceSlot = e.Slots[0];
    }
    ...
}
```

Listing 11. Retrieval of slot in extension template.

4.2 Discovering templates in Plux

To generate extensions from templates essentially means to replace metadata placeholders with real metadata. In the Plux CI, metadata is retrieved by the discovery core. Thus, support for templates can be implemented by a special discovery extension called *TemplateAnalyzer*.

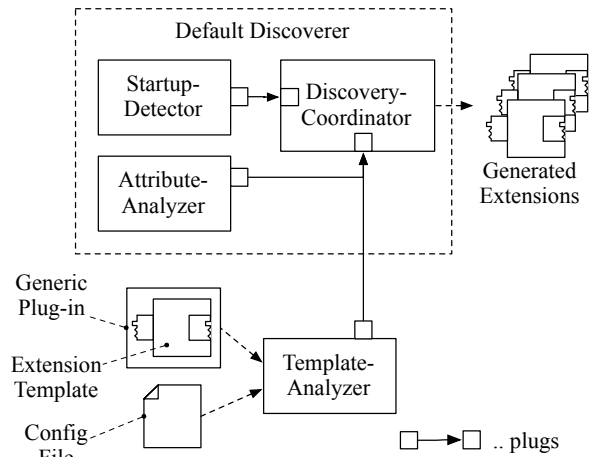


Figure 12. Integration of the the template analyzer into the discovery mechanism.

Figure 12 shows the template analyzer and how it integrates with the discovery coordinator. After a detector has found a plug-in, the discovery coordinator directs the ana-

lyzers to search its metadata. The attribute analyzer ignores the generic plug-in *Grid.dll*, after it unsuccessfully checked for the [Extension] attribute. If the template analyzer inspects *Grid.dll*, it finds the [Template] attribute and therefore knows that this is a plug-in from which it can extract metadata. The template analyzer then searches the configuration file for metadata matching the plug-in file and template name (cf. *Grid.dll/Grid* in Listing 9). It generates an extension meta element with the name *ArticleGrid* and reads the plug, param, and slot attributes from the template. For this purpose, it uses the attribute analyzer. Finally, it replaces the placeholders with the metadata from the configuration file and passes the created metadata to the discovery core.

5. Case Study

To validate Plux and generic plug-ins, we are conducting a case study with our industrial partner BMD Systemhaus GmbH. BMD is a medium-sized company offering enterprise software products to 18.400 customers and 45.000 active users mainly in Austria, Germany, and Hungary. BMD Software is a comprehensive suite of enterprise applications for customer relationship management (CRM), accounting, cost accounting, payroll, enterprise resource planning, as well as for 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.

To pursue customizable products, BMD initiated a pilot project where we applied Plux to the CRM product. We have developed a set of usage scenarios demonstrating the need for a reconfigurable application with support for dynamic addition and removal of features [9]. In BMD's market environment, customization enables two major scenarios: (1) Customize the feature-rich enterprise application for individual users by assembling custom applications from plug-ins that BMD offers with its core product. (2) Customers should be able to contribute their custom extensions, because even if the core product covers all the major business-relevant scenarios, customers typically ask for more features addressing their special needs.

Building the enterprise application with the Plux infrastructure enables Scenario 1. We partly ported the CRM application to Plux to demonstrate customized applications [9][10][11]. The further work for Scenario 2 led to generic plug-ins. The motivating example from Section 3 covers the most frequent extensibility scenario: A third-party wants to contribute a custom view. When third-parties contribute functionality, their effort is significantly reduced if they use templates, because templates allow them to reuse general-purpose extensions. For example: a standard grid can be customized as a data grid, a customer grid, or an article grid; a standard filter panel can be customized to filter customers, articles, or employees; a standard menu bar or tool bar can be customized for different views. For data

integration, controls provided by the third party integrate BMD's data source extensions.

6. Related Work

In programming languages, generic data types are a well-known concept that was pioneered by Ada in the early eighties and is now part of most modern languages [12]. Generic programming allows developers to define abstract data types that can be parameterized by other types which are specified later on. It reduces code duplication and promotes type-safety.

The implementation of generic data types in mainstream languages such as Java, C#, and C++ differs [13][14][15]. In C++, generic types are described by templates. For each specific usage of a template the C++ compiler generates a new type. Thus, at run time, the generated types are like regular types. In C#, genericity is a concept that is not only supported by the language but also by the virtual machine (the Common Language Runtime). At run time, the types are still generic. However, the just-in-time compiler closes the types, i.e., the type parameters are replaced with specific types. In comparison to C#, Java generics are simpler, because Java supports genericity only at source code level, but not at the machine level. In other words, it does not generate types at all. Instead, Java uses *Object* references at run time and does all type checking in the java source compiler.

In plug-in systems, generics have not been an issue so far. Plug-in frameworks such as OSGi or Eclipse rely on programmatic composition, where programmers connect components explicitly. These frameworks can of course make use of generic classes, but generic metadata are not necessary because there is no automatic composition where the same component should be used with different metadata at various places in the code. In Plux, we have automatic composition guided by metadata. Since the metadata control which components get connected, they must be different for each reuse. Generics are a means to solve this problem by customizing the metadata of reusable components for different contexts.

In OSGi [3], contributors register in a global service registry and hosts look up services in this registry. If a host connects to a contributor, it cannot control which services the contributor will use. If such control is required, the contributor must provide a custom configuration interface.

In Eclipse [4], extensions live in a global registry and are discovered based on XML configuration files. The metadata in the configuration files specify to which extension points (i.e., slots) the extension contributes. Providing multiple XML files with different metadata, allows generating multiple instances of an extension. However, if the generated extension has extension points itself, Eclipse does not generate their names from XML metadata. Instead, the names of extension points are hard-coded in the extension. In Plux, the names of slots and plugs are part of the metadata and can be customized for generic extensions.

7. Conclusions

In this paper we presented an approach for generic plug-ins and their integration into the Plux plug-in framework. Generic plug-ins solve the problem of reusing general-purpose components in the context of automatic composition, where the components declare their requirements and provisions, and a composer inside the infrastructure composes an application by matching those requirements and provisions. The composer depends on metadata that specify which components should be connected.

Generic plug-ins contain general-purpose reusable extensions as templates. The templates use placeholders for metadata that direct the composer. At run time, when the infrastructure discovers the templates, their placeholders are substituted by concrete metadata from external sources.

Together with our industry partner BMD we have shown the feasibility and usefulness of generic plug-ins in a case study. Our approach allows generating custom extensions from BMD's templates and external composition metadata.

In future work we will develop coordination mechanisms for extensions generated from templates. We learned in the case study, that if we generate extensions from templates, and if we want to compose them without programming, we also want to define relationships between them just by specifying composition metadata. So far, we can only define simple relationships, e.g., we can use a shared data contributor to share a current data record or share a certain filtering mechanism. However, we cannot combine a current record in one data contributor with the filtering of another data contributor. This is necessary for example in a master-detail relationship. A master-detail relationship between two extensions means that a *detail* part contains detailed information which is associated to the current selection in the *master* part.

With master-detail relationships, we want to automatically compose data contributors guided by metadata in three ways: (1) A master can control multiple details. For example, in a user administrator, a list of groups (master) can show associated users (first detail) and associated group privileges (second detail) for the selected group. (2) An extension can be both master and detail. For example, a list of groups (master) can show the associated users (detail) for the selected group. Vice versa, a list of users (master) can show a list of groups (detail) to which the selected user belongs. (3) Masters and details can be chained. For example, a list of groups (master) can show the associated users (here as detail) for the selected group. In a chain, the list of users (here as master) shows the groups to which the user belongs (detail).

8. References

- [1] Wolfinger, R.: Dynamic Application Composition with Plux.NET: Composition Model, Composition Infrastructure. Dissertation, Johannes Kepler University, Linz, Austria, 2010.
- [2] Birsan, D.: On Plug-ins and Extensible Architectures. *ACM Queue*, 3(2):40–46, 2005.
- [3] OSGi Service Platform, Release 4. The Open Services Gateway Initiative, <http://www.osgi.org>, July 2006.
- [4] Eclipse Platform Technical Overview. Object Technology International, Inc., <http://www.eclipse.org>, February 2003.
- [5] Boudreau, T., Tulach, J., Wielenga, G.: Rich Client Programming, Plugging into the NetBeans Platform, 2007.
- [6] Microsoft: Microsoft C# Language Specifications. Microsoft Press, Redmond, 2001.
- [7] Jahn, M., Löberbauer, M., Wolfinger, R., Mössenböck, H.: Rule-based Composition Behaviors in Dynamic Plug-in Systems. Submitted to The 17th Asia-Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30-December 3, 2010.
- [8] Jahn, M., Wolfinger, R., Mössenböck, H.: Extending Web Applications with Client and Server Plug-ins. *Software Engineering 2010, SE 2010*, Paderborn, Germany, February 22-26, 2010.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] Barnes, J.: *Programming in Ada 95*. Addison-Wesley Longman, Amsterdam, 2006.
- [13] Sun Microsystems: Java Platform, Standard Edition 6, API Specification. <http://java.sun.com/javase/6/docs>, 2006.
- [14] Richter, J.: *CLR via C#. Applied Microsoft .NET Framework 2.0 Programming*. Second Edition, Microsoft Press, 2006.
- [15] Stroustrup, B.: *The C++ Programming Language*. Addison-Wesley Longman, Amsterdam, 2000.