Adding genericity to a plug-in framework

Florian Oeser (s780586)

Beuth Hochschule für Technik Berlin,
Luxemburger Straße 10, 13353 Berlin, Germany
florian.oeser@gmail.com
http://www.florian-oeser.de/2012/10/04/wissenschaftliches-arbeiten/

Abstract. Diese Ausarbeitung wurde auf Basis des Papers *Adding genericity to a plug-in framework* [1] verfasst. Bevor der Inhalt des Papers zusammenfassend erläutert wird, werden im ersten Teil der Ausarbeitung für das Verständnis notwendige Grundlagen und Konzepte beschrieben. Abschließend wird das vorgestellte Paper mit anderen Arbeiten verglichen und bewertet.

1 Plug-In's in der Softwareentwicklung

Der Grund für eine Plug-In-basierte Entwicklung ist Programme flexibler zu gestalten. Im Bereich der Softwareentwicklung spricht man von einem Plug-in, wenn eine oder mehrere Softwarekomponenten eine bestehende Anwendung um eine bestimmte Funktionalität erweitern. Dies geschieht dynamisch während der Laufzeit und somit ohne Neustart der Hostanwendung [2][3].

Anwendungen sind oft monolithisch und schwergewichtig. Das kann bedeuten, das kleine Änderungen es erfordern, dass die ganze Software neu ausgeliefert werden muss. Dazu kommt, dass ein Benutzer meist nur Bruchteile der bereitgestellten Funktionalität benötigt. Der Plug-in-Ansatz erlaubt es, Programme in Teile zu zerlegen, die dann von Endbenutzern, je nach Bedarf, zu unterschiedlichen Programmkonfigurationen zusammengesetzt werden können [3]. Abbildung 1 verdeutlicht das anhand einer Gegenüberstellung. Das Zusammensetzten unterschiedlicher Programmteile ermöglicht es, Anwendungen in ihrer Größe, Komplexität und Kosten besser zu skalieren [2]. Zudem erhalten die Nutzer, aber auch Drittentwickler, die Möglichkeit Programme um neue Funktionalitäten zu erweitern und an ihre Bedürfnisse anzupassen.

Zusammenfassend werden folgende Ziele für einen Plug-in-basierten Ansatz definiert:

- Dynamisches Hinzufügen und Entfernen von Plug-ins ohne Neustart der Applikation
- Benutzer läd nur das, was er braucht
- Einfache Erweiterbarkeit (Plug & Play) ohne programmieren oder konfigurieren
- Sicherheitsrichtlinien (Wer kann ein Plug-in hinzufügen? Was darf ein Plug-in?)

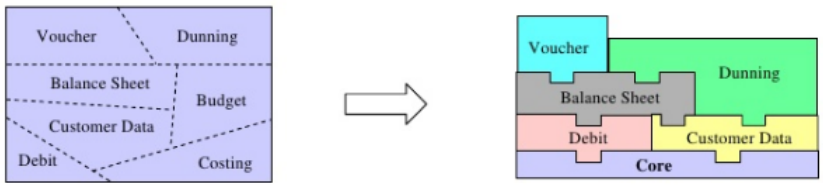

Fig. 1. Software-Monolith gegenüber einem schlanken Kern plus Plug-ins

In dieser Arbeit wird ein Plug-in Framework vorgestellt, welche diese Zielsetzungen erfüllt. Dabei beruht die Architektur auf .NET Konzepten wie Attributen und Metadaten, um relevante Informationen zu einem Plug-in direkt im Sourcecode einer Anwendung zu spezifizieren. In [4] argumentieren die Autoren, das dieser Ansatz einfacher zu lesen und zu warten ist, als beispielsweise die Plug-in Architektur von Eclipse.

1.1 Plug-in Architektur

Plugin-in Architekturen werden häufig über Interfaces realisiert, da sie eine Art Vertrag zwischen der Hostanwendung und dem Plug-in ermöglichen. Jede Klasse die ein Interface implementiert, muss für jede Methode des Interface eine Implementierung bereitstellen. Somit weiß jede Anwendung, die das Interface kennt, welche Methoden durch ein Plug-in implementiert wurden und welche Methoden sie aufrufen kann [2].

Ein essentielles Prinzip einer jeden Plug-in Architektur ist das *Plugging* [4]. Das System hat sicher zu stellen, das Erweiterungen in einer kontrollierten, eingeschränkten und festgelegten Art und Weise verwendet werden. Eine Plug-in Architektur muss also über Mittel verfügen, welche festlegen, wie Komponenten erweitert werden können und wie andere Plug-in's ihre Funktionalität bereitstellen [3]. Dadurch ist definiert, welche Komponenten zusammengefügt - *geplugged* - werden können.

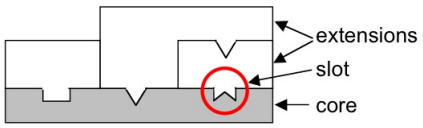

Fig. 2. Eine Erweiterung (hier *extension*) plugged in den Slot der Anwendung (*core*)

Wie Abbildung 2 vedeutlicht, können eine Reihe von Erweiterungen (*Extensions*) in die für sie vorgesehenen *Slots* zur Laufzeit geplugged werden. Ein Slot definiert ein Interface und eine Liste von Parametern mit Namen und Typen. Eine Extension liefert die passende Implementierung sowie eine Liste von Werten

für diese Parameter. Der in dieser Arbeit vorgestellte Ansatz verwendet ähnliche Begrifflichkeiten, auf welche im nächsten Kapitel näher eingegangen wird.

Plug-in Komponenten können als kleine Anwendungen gesehen werden, welche eine Host-Anwendung um neue Dienste erweitern. Nichttriviale Dienste können aus mehreren Erweiterungen bestehen, welche an unterschiedlichen Stellen in die Anwendung geplugged werden. Erweiterungen, welche logisch zusammen gehören, werden in einer Komponente ausgeliefert. Man spricht dabei auch von *Deployment* [4].

Der Begriff *Discovery* bezeichnet das automatische Erkennen und Aktivieren von Plug-in's zur Lade- und Laufzeit einer Applikation [4][3]. Dieser Vorgang sollte sicher und einfach gestaltet sein und ohne fehleranfällige Konfigurationsaufgaben einhergehen. Der in dieser Arbeit vorgestellte Ansatz verwaltet Plug-in's in einem zentralen *plugin-in repository* als Teil einer vorgegebenen Verzeichnisstruktur.

1.2 Generizität

Generische Datentypen sind in der Programmierung ein altbekanntes Konzept und werden durch viele moderne Programmiersprachen unterstützt [1]. Generische Programmierung erlaubt es Entwicklern abstrakte Datentypen zu deklarieren, welche durch andere Typen parametrisiert werden können und später, zur Laufzeit, spezifiziert werden. Dies reduziert die unnötige Vervielfältigung von gleichem Code und fördert Typsicherheit. In C++ werden generische Typen als Templates beschrieben. Für jede spezifische Verwendung des Templates generiert der C++ Compiler einen neuen Typen. Somit lassen sich die generierten Typen nicht von den normal deklarierten unterscheiden. In C# ist das Konzept der Generizität nicht nur Teil der Sprache, sondern auch der Laufzeitumgebung (*CLR*). Somit sind die Typen zur Laufzeit weiterhin generisch. Allerdings werden sie vom *just-in-time*-Compiler geschlossen und die Typparameter werden mit ihren spezifischen Typen ersetzt. In Java exisitiert Generizität nur im Sourcecode, nicht aber auf Maschinensprachen-Level. Dadurch gibt es in Java keine generierten Typen. Java (*JRE*) benutzt zur Laufzeit *Object*-Referenzen und überlässt die Typüberprüfung allein dem Java-Compiler [1].

Generizität wird in Plug-in Systemen notwendig, wenn allgemein gehaltene Komponenten (als Plug-in's implementiert) wiederverwendet werden sollen. Komponenten dieser Art sind Grid's, die in unterschiedlichen View's, unterschiedliche Datensätze anzeigen sollen. Dadurch müssen die Informationen, mit Hilfe derer die Plug-ins in dem hier präsentierten Plug-in System zusammengefügt werden, flexibel sein. Erreicht wird das, indem diese Informationen über einen Template-basierten Ansatz erst zur Laufzeit konkretisiert (typisiert) werden.

2 .NET-Framework Konzepte

Das .NET-Framework stellt technische Grundlagen zur Verfügung, auf dem der im nächsten Kapitel beschriebene Plug-in Ansatz basiert. Folgend werden die

benötigten Konzepte der .NET *Attribute*, *Assemblies*, *Metadaten* und *Reflexion* kurz vorgestellt.

2.1 Reflexion

Reflexion ermöglicht es, die Struktur (Namespaces, Klassen etc.) eines Programmes zur Laufzeit abzufragen und zu modifizieren [5]. Bei Methoden werden beispielsweise Information über die Sichtbarkeit oder die Datentypen von Übergabeparametern bereitgestellt. Für die Realisierung von Reflexion ist das Speichern von Metainformationen[1] im Binärcode des Programms notwendig. .NET unterstützt Reflexion dahingehend, dass alle Sprachen, die das .NET-Framework verwenden, automatisch die entsprechenden Informationen als Metadaten speichern.

```
public String GetStringProperty(Object obj, String methodName)
{
    String value = null;
    try  {
         MethodInfo methodInfo =
              obj.GetType().GetMethod(methodName);
         value = (String)methodInfo.Invoke(obj, new Object[0]);
    } catch (Exception e) {}
    //Fehlerbehandlung zwecks Übersichtlichkeit nicht
         implementiert.

    return value;
}
```

Listing 1.1. `GetStringProperty()` liefert über Reflexion den Rückgabewert einer Methode eines gegebenen Objektes

Listing 1.1 zeigt eine Methode, die eine beliebige andere Methode eines gegebenen Objektes aufruft und deren Rückgabewert zurückliefert.

2.2 Attribute

Attribute sind Meta-Informationen, welche im Sourcecode an Sprachkonstrukte wie Klassen, Interfaces, Methoden und Felder angefügt werden. Sie sind das Pendant zu den aus Java bekannten *Annotations*. Zur Laufzeit können diese Attribute via Reflexion ausgelesen werden. Zusätzlich zu Attributen, die in der Basisklassenbibliothek der CLR[2] definiert sind, ist es möglich, benutzerdefinierte Attribute zu erstellen, um dem Code ergänzende Informationen hinzuzufügen [4].

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

Listing 1.2. Einer Klasse wird ein Attribut zugewiesen

[1] Enthalten Information über Merkmale anderer Daten
[2] Common Runtime Language; Laufzeitumgebung von .NET

Listing 1.2 zeigt, wie der Klasse `StockTicker` das `[WebMethod]`-Attribut hinzugefügt wurde. Dieses Attribut gibt an, dass die Methode als Teil eines *XML Web services* exponiert wird. Über Reflexion kann dieses Attribut ausgelesen werden und die Methode vom Client-Code entsprechend behandelt werden.

Das in dieser Ausarbeitung behandelte Plug-in Framework benutzt Attribute um Informationen zu Plug-in-Komponenten zu deklarieren.

2.3 Assemblies

Ein Assembly ist der Grundbaustein einer jeden .NET Framework-Anwendung. Es ist die kleinste Basiseinheit für das Laden, das Deployment, der Wiederverwendung, der Versionskontrolle und der Sicherheitsberechtigungen. Assemblies werden als ausführbare Dateien (*.exe) oder Bibliotheken (*.dll) ausgeliefert. Sie beinhalten Metadaten, welche Typen beschreiben, Ressourcen und referenzierte Assemblies [4]. Nach außen bilden sie eine logische, funktionelle Einheit. Eine Assembly stellt der CLR die für das Erkennen von Typimplementierungen erforderlichen Informationen zur Verfügung. Für die CLR sind Typen nur im Kontext einer Assembly vorhanden.

Assemblies können signiert werden. Dies wird als das Vergeben eines starken Namens bezeichnet. Mit Hilfe dieses starken Namens und einer Versionsinformation eines Assemblies können später Komponenten identifiziert werden. Ein Beispiel einer deklarativen Versionierung zeigt Listing 1.3. Die so durch ein Attribut markierte Version lässt sich später per Reflexion auslesen.

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

Listing 1.3. Versionierung eines Assemblies

In dem vorgestellten Plug-in Framework werden Assemblies als Container für Plug-in Komponenten verwendet. Die Signatur und die Versioninformation wird dabei genutzt, um Plug-in's zu identifizieren.

2.4 Metadaten

Ein Assembly speichert neben Code auch Metadaten, welche die Symbolinformationen aller Typen, Methoden und Felder in diesem beschreiben. Die Metadaten werden automatisch vom Compiler aus dem Sourcecode erzeugt. Hier ermöglicht es .NET die Metadaten via Reflexion zur Laufzeit aus dem Assembly auszulesen [4]. Listing 1.4 zeigt exemplarisch wie nach allen Methoden der `StockTicker`-Klasse gesucht wird, welchen das `[WebMethod]`-Attribut hinzugefügt wurde.

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

```
            // use WebMethodAttribute
          }
        }
```

Listing 1.4. Reflexion erlaubt das Auslesen von Metadaten aus Assemblies zur Laufzeit

Das in dieser Ausarbeitung erörterte Plug-in Framework nutzt Reflexion zum Erkennen von Plug-in's. Dabei wird eine spezielle Verzeichnisstuktur durchsucht, wobei die Definitionen gefundener Plug-in's aus den Metadaten gelesen werden.

3 Plux.NET

In [1] stellen die Autoren ein Plug-in Framework für Microsoft .NET vor. Dieses Plug-in Framework ermöglicht Software-Kompositionen durch ein *Plug and Play*-Mechanismus und wird als *Plux.NET* bezeichnet.

Plux.NET definiert ein Kompositionsmodell bzw. Komponentenmodell und führt die *Slot und Plug*-Metapher ein. Ein Komponentenmodell legt nach [6] einen Rahmen für die Entwicklung und Ausführung von Komponenten fest. Es werden strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten dieser definiert.

Ein Slot und ein Plug verschmelzen zu einer *Extension*. Eine Extension lässt sich als Komponente betrachten, welche zum einen Funktionalitäten bereitstellt (dann als Slot bezeichnet), aber auch Funktionalitäten von anderen Extensions beziehen kann (dann als Plug bezeichnet). Dies wird durch Abbildung 3 verdeutlicht. Man spricht von *host extensions* bzw. *Host* oder *contributor extensions* bzw. *Contributor*, jenachdem ob eine Extensionen einen Slot öffnet oder einen Plug bereitstellt.

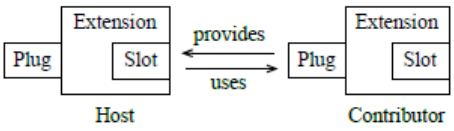

Fig. 3. Slot-Plug-Beziehung von zwei Extensions

Ein Slot spezifiziert ein Vertrag über ein Interface. Eine Contributor bzw. ein Plug muss dafür eine Implementierung bereitstellen. Zusätzlich kann ein Slot eine Liste von Parametern mit Namen und Typangaben definieren, wobei ein Plug konkrete Parameterwerte liefern muss. Dies geschieht (standardmäßig[3]) deklarativ über Metadaten (.NET-Attribute). Diese Metadaten werden im Sourcecode an Sprachkonstrukte wie Klassen, Interfaces oder Methoden gebunden und werden zur Laufzeit über Reflexion ausgelesen. Der Slot und der Plug werden über

[3] Plux.NET erlaubt es wie in Eclipse, Metadaten über externe Konfigurationsdateien zu definieren

eindeutige Namen identifiziert. Dies geschieht ebenfalls über ein Metaelement (`[Slotdefinition("name")]` und `[Plug("name")]`). Ein Plug passt zu einem Slot, wenn ihre Namen übereinstimmen.

Listing 1.5 und 1.6 verdeutlichen diesen Sachverhalt. Es wird ein Host bzw. ein Slot mit dem Namen `Logger` definiert, welcher Lognachrichten mit einem Timestamp ausgibt. Dazu wird das `[SlotDefinition]`Attribut verwendet. Eine konkrete Implementierung findet im Plug (Contributor), definiert durch das `[Plug]`-Attribut, statt. Dazu wird das Interface des Slots implementiert und dessen eindeutiger Name (`Logger`) über die Metadaten deklariert. Ein konkretes Zeitformat (`hh:mm:ss`) wird über den Parameterwert festgelegt.

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

Listing 1.5. Definition eines Logger-Slots

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

Listing 1.6. Definition eines passenden Plug's für den Logger-Slot (Konsolenausgabe)

Das Plux.NET Framework selbst ist Plug-in basiert. Das bedeutet, dass die eigentliche Anwendung ebenfalls als Plug-in implementiert werden muss. Das Framework definiert einen entsprechenden Slot (`Application`), welcher mit einem passenden Plug in der eigenen Anwendung implementiert wird (siehe Listing 1.7). Die Anwendung hat weiterhin einen Slot `Logger`, welcher bereits wie in Listing 1.6 gezeigt implementiert wurde und die eigentliche Funktionalität bereitstellt.

```
[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 1.7. Applikation mit Plug zum Framework und Slot zum Logger-Plug

Der Anwendung wird im Konstruktor beim Laden durch das Framework ein Objekt mit den Metadaten der assoziierten Extensions übergeben, aus dem eine Referenz auf den `Logger`-Slot geholt wird. In einem seperaten Thread werden dann von allen Plugs, die den `Logger`-Slot implementieren, die `Print()`-Methode aufgerufen. In diesem Beispiel die des `ConsoleLogger`-Plugs, definiert in Listing 1.6. Dies ist ein dynamischer Prozess, da unterschiedliche `Logger`-Implementierungen zur Laufzeit hinzugefügt bzw. entfernt werden können. Dazu wird vom Kompositionsmodell die `PluggedPlugs`-Collection des `Logger-Slots` geupdatet und die Anwendung kann entsprechend reagieren.

Der Slot beziehungsweise das Interface `ILogger` (Listing 1.5), die Plugs bzw. die Klassen `ConsoleLogger` (Listing 1.6) und `MyApp` (Listing 1.7) werden zu DLL's (Assemblies) kompiliert und in die Plux.NET Plug-in-Verzeichnisstruktur kopiert (Deployment). Das Framework erkennt die Extension `MyApp` und fügt diesem dem eigenen `Application`-Slot hinzu. Weiterhin erkennt das Framework die `ConsoleLogger`-Extension und fügt diese dem `Logger`-Slot der `MyApp`-Anwendung hinzu (Discovery).

Zusammenfassend besitzt Plux.NET ein Kompositionsmodell, welches eine Anwendung aus Komponenten bzw. Extensions zusammenfügt, welche wiederum aus einer definierten Anforderung (Slot) und einer passenden Implementierung (Plug) bestehen. Das Framework verbindet diese automatisch anhand der deklarierten Metadaten, wobei diese per Reflection aus den Assemblies extrahiert werden. Das Kompositionsmodell sucht Extensions (Assemblies) in einer definierten Verzeichnisstruktur. Im Gegensatz zu anderen Plug-in-Systemen ist das Extrahieren der Meta-Informationen, sowie die Suche nach den Extensions, nicht integraler Bestandteil des Frameworks. Es ist Plug-in basiert und lässt sich beliebig austauschen. So könnten Extensions über das Netzwerk statt über das Dateisystem gefunden werden oder die Metadaten aus einem XML-File gelesen werden, statt über .NET-Attribute.

Weitere, hier nicht diskutierte Features von Plux.NET sind das Rechte- und Sicherheitssystem welches beispielsweise festlegt, welche Extension welchen Slot öffnen darf, bzw. welche Extensions in einen Slot pluggen dürfen. Ebenfalls nicht diskutiert werden die *slot behaviors* (spezifizieren Kompositionsverhalten von Slots) und die *scripting API*, welche es erlaubt, bestimmte Operationen des Kompositionsmodells zu überschreiben.

3.1 Generische Plug-Ins

Die Metadaten der Slots und Plugs definieren, welche Host- bzw. Contributor-Extensions zusammen passen. Dadurch ist es nicht so ohne weiteres möglich allgemein gehaltene Extensions bzw. Komponenten wiederzuverwenden. Es müssten jedesmal neue Metadaten definiert und unterschiedlich ausgeprägt werden.

Abbildung 4 zeigt eine Applikation mit zwei Views. Die eine View zeigt in einem Grid Kundendaten an und die andere View zeigt Artikeldaten in einem anderen Grid an. Jede View besitzt ein Filter-Panel mit den die Datensätze sortiert und durchsucht werden können. Eine entsprechende Extension für den View (als Host) würde einen Slot öffnen, an den Controls pluggen könnten. In dem konkreten Fall würden zwei Contributor's (Plug's) für diesen Slot definiert werden. Einer für das Grid-Control und einer für das Filter-Panel-Control (vgl. Abbildung 5). Beide benötigen, um Daten anzeigen und sortieren zu können wiederum einen Slot in den der eigentliche Daten-Contributor plugged. Abbildung 5 verdeutlicht dies schematisch. Listing 1.8 zeigt die Slot-Definition für die Datenquelle und das Control anhand von Quellcode.

```
[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 1.8. Slot-Definition für ein Control und die Datenquelle

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

Listing 1.9. Metadata für die Grid-Extension

Das Grid- und das Filter-Panel sind als allgemein gehaltene Extensions zu verstehen, welche an verschiedenen Stellen der Applikation verwendet werden könnten. Beispielsweise könnte das Grid in einer ganz anderen View, statt Daten, Objektproperties anzeigen. In dem bisherigen Beispiel werden sie nur zweimal instantiert, jeweils für einen View (Artikel- und Kunden-View). Wie in Listing 1.9 und Abbildung 5 schematisch zu sehen, hat die `Grid`-Extension ein `Control`-Plug für die View und ein `DataSource`-Slot für die Bereitstellung der Daten. Die Problematik die hierbei entsteht ist, dass wenn das Framework die

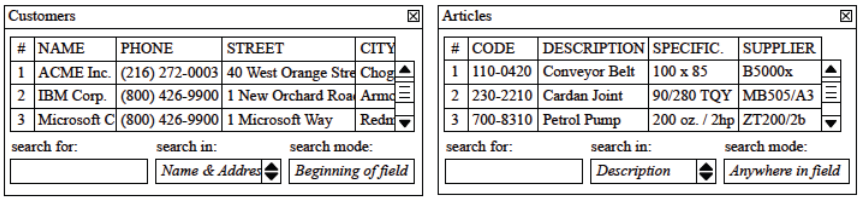

Fig. 4. Artikel- und Kunden-View als beispielhafte UI

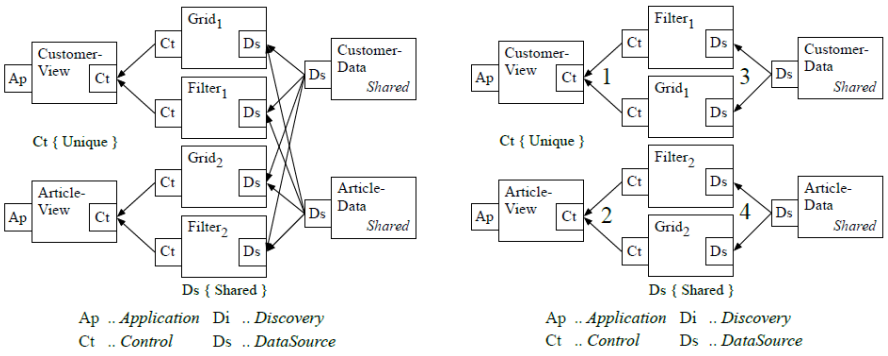

Fig. 5. Inkorrekte (links) und korrekte (rechts) Komposition

`DataSource`-Slots füllen will, jeden Daten-Contributor (also die Datenquelle) in jedes Grid plugged, da jedes Plug eines Daten-Contributors zu den Slots aller Grids passt. Die Datenquelle für Kundendaten plugged in das Grid für den Kunden-View sowie in das für den Artikel-View. Das betrifft gleichermaßen die Filter-Extension. Dies wird durch das linke Schema in Abbildung 5 verdeutlicht. Eine weitere Problematik ist, dass das Framework jedes `Control`-Plug eines Filter oder Grids in jeden View plugged. Es ist nicht kontrollierbar, das ein bestimmtes Control nur an eine bestimmte View geplugged wird. Beispielsweise wenn das Filter-Panel nur im Artikel-View und nicht im Kunden-View verwendet werden sollte. Das rechte Schema in Abbildung 5 zeigt eine valide Komposition. Die `CustomerData`-Extension (3) plugged nur in Controls, welche einen Plug zu einem Kunden-View (1) haben und die `ArticleData`-Extension (4) plugged nur in Controls, welche einen Plug für einen Artikel-View (2) bereitstellen.

Für eine korrekte Komposition eines Artikel- und Kunden-Views müssten die Controls mit den Views und die Datenquellen mit den Controls selektiv verbunden werden. Ohne einen generischen Ansatz stehen zwei Lösungswege zur Verfügung.

Es kann der Komposer deaktiviert werden und die Extensions werden programmatisch verbunden. Das widerspricht der Idee von Plug-ins, ohne großen Programmieraufwand Software flexibler zu gestalten. Weiterhin würde eine programmatische Komposition von Controls dazu führen, das eine entsprechende

View nicht mehr erweitert werden kann. Eine View kann keine Controls integrieren, welche nicht zur Übersetzungszeit bekannt waren. Hinzu kommt, dass wenn alle Datenquellen den selben Plug-Name nutzen, andere Hosts dann ebenfalls manuell zusammengesetzt werden müssten, wenn diese auch die Datenquelle benutzen wollen.

Mit Hilfe unterschiedlich ausgeprägter Metadaten könnten ebenfalls korrekte Kompositionen von Artikel- und Kunden-Views erreicht werden. Durch verschiedene Metadaten würden dem jeweiligen Kontext (Artikel- oder Kunden-View) entsprechende Grid- und Filterextensions entstehen. Dabei werden für jede Wiederverwendung neue Extensions (Subklassen) erzeugt (siehe Listing 1.10). Beispielsweise könnte eine `ArtikelGrid`-Klasse von der bereits bekannten `Grid`-Klasse (Listing 1.9) erben und einen konkret ausgeprägten Slot für die Datenquelle (`ArticleData)`, einen spezifischen Plug (`ArticleControl`) für das Control und verschiedene Parameterwerte implementieren.

```
[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 1.10. Wiederverwendung von Extensions über sub-classing

Dieser Ansatz erlaubt automatische Kompositionen, erzeugt aber viele unnötige Typen, da bei jeder Wiederverwendung neue Klassen angelegt werden, obwohl lediglich neue Metadaten benötigt werden würden.

Ein generischer Ansatz sollte es erlauben, Extensions mit Metadaten zu generieren, welche unabhängig von der implementierenden Klasse sind. Weiterhin sollte es möglich sein, mehrere Extensions mit unterschiedlichen Metadaten aus einer einzigen Klasse zu generieren.

Der generisches Plug-in in Plux.NET beinhaltet ein oder mehrere *extension templates*. Ein extension template enthält eine Klasse, Metadaten welche den Komposer konfigurieren und Platzhalter für die bereits bekannten Metadaten wie Slot-Namen, Plug-Namen und Parameterwerte. Im Gegensatz zu regulären Extensions sind Templates, bedingt durch die mit Platzhaltern bestückten Medadaten, unvollständig. Um Extensions aus Templates zu generieren, muss das Framework die Platzhalter der Metadaten aus einer externen Quelle, wie einer Konfigurationsdatei oder Datenbank, ersetzen.

Im Folgenden wird das obrige Beispiel, mit zwei verschiedenen Views, unter dem Gesichtspunkt des generischen Ansatzes betrachtet.

Wie in Abbildung 6 zu sehen hat das Template `GridTemplate` vier Platzhalter. Den `<Grid>`-Platzhalter für den Extension-Name, den `<Control>`-Platzhalter für den Plug-Namen, den `<DataSource>`-Platzhalter für den Slot-Name und den `<Order>`-Platzhalter für einen Parameterwert. Die externen Metadaten (vgl. Abbildung 6) spezifizieren, dass zwei Extensions aus dem Template generiert werden. Eine `ArticleGrid`-Extension und eine `CustomerGrid`-Extension. In der `ArticleGrid`-Extension ersetzt das `ArticleControl`-Plug den `<Control>`-

Platzhalter, der `ArticleData`-Slot ersetzt den `<DataSource>`-Platzhalter und der float-Wert von 0.5 wird für den `<Order>`-Platzhalter eingesetzt. Die Platzhalter in der `CustomerGrid`-Extension werden in gleicher Weise ersetzt.

Listing 1.11 zeigt das `Grid` als Template-Definition. Um ein Template im Code zu deklarieren, wird das `Template`-Attribute verwendet. Für die Plug-, Slot- und Parameter-Definition werden die bereits bekannten Attribute verwendet. Um Platzhalter von finalen Metadaten zu unterscheiden, werden die Platzhalternamen in spitze Klammern gesetzt. Über den konkreten Namen der Platzhalter lassen sich diese nicht nur aus den externen Metadaten ersetzten, sie spezifizieren ebenfalls die Slot-Definition auf welchen ein Slot oder Plug (namentlich) basiert. Beispielsweise basiert der Template-Slot `<DataSource>` auf der Slot-Definition `DataSource` wie in Listing 1.8 zu sehen.

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

Listing 1.11. Definition eines Grid-Templates

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

Listing 1.12. (Article-)Grid-Extension generiert aus einem Template und externen Metadaten

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

Listing 1.13. Externe Metadaten für das Grid-Template

Listing 1.12 zeigt die finale `Grid`-Klasse als Resultat aus der Template-Definition (Listing 1.11) und den externen Metadaten (Listing 1.13).

Wurden bisher Slots über ihre Namen, aus dem durch das Framework im Konstruktor übergebene Metadaten-Objekt der assoziierten Extensions extrahiert und referenziert (vgl. Listing 1.7), werden für den template-basierten Ansatz Indizies verwendet. Das hat den Grund das in diesem Falle die konkreten Namen nicht zur Übersetzungszeit bekannt sind. Um auf den Slot einer Extension im Code zuzugreifen, wird ein Slotindex statt ein Slotname verwendet. Dies verdeutlicht der Sourcecode in Listing 1.14.

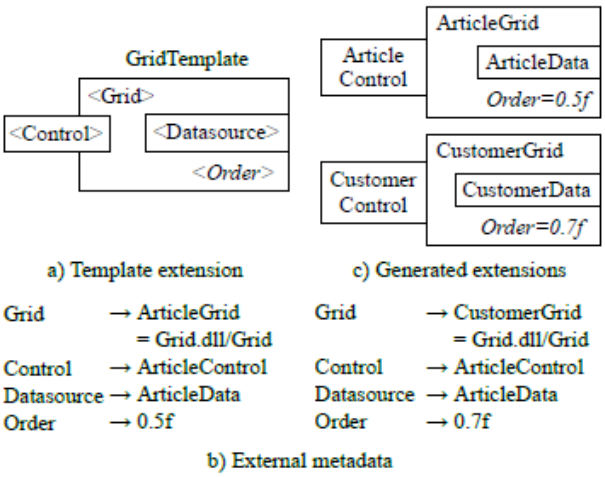

Fig. 6. Grid-Extensions generiert aus einem Template und externen Metadaten

```
[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 1.14. Slot-Referenzierung über Indizes statt Namen

Das Generieren von Extensions aus Templates bedeutet das Austauschen von Metadaten-Platzhaltern mit echten Metadaten. Dies wird wie bisher durch die Kompositionsinfrastrukur des Frameworks übernommen. Bleibt die Suche nach dem `[Extension]`-Attribut in einem Assembly erfolglos (beispielsweise für das Grid-Template in Listing 1.11), untersucht ein spezieller *template analyzer* das Assembly nach dem `[Template]`-Attribut. Wird dies gefunden, sucht der template analyzer nach einer Konfigurationsdatei mit Metadaten, wobei der Template-Name und der Plug-in Assembly-Name übereinstimmen müssen (siehe Metadaten in Abbildung 1.9; `Grid.dll/Grid`). Das Framework generiert daraus eine Extension (`ArticleGrid`) und liest die Plug-, Slot- und Parameter-Attribute und ersetzt die Platzhalter mit den Metadaten aus der Konfigurationsdatei.

4 Zusammenfassung und Vergleich

In [1] wurde ein Ansatz für generische Plug-ins vorgestellt. Ebenfalls wurde eine Integration in das Plux.NET Plug-in Framework vorgenommen. Plux.NET

ist ein leichtgewichtiges Plug-in Framework für .NET. Es ermöglicht deklarative Kompositionen von Komponenten indem relevante Informationen durch Attribute und Metadaten direkt im Sourcecode einer Anwendung spezifiziert werden. Zur Laufzeit werden diese Informationen per Reflexion ausgelesen.

Generische Plug-in's lösen das Problem der Wiederverwendung von allgemein gehaltenen Komponenten im Kontext von automatischen Kompositionen. Dabei deklarieren Komponenten ihre Anforderungen und Vorschriften über Metadaten wobei das Framework Anwendungen anhand dieser zusammenfügt. Wiederverwendbare Extensions werden in generischen Plug-ins als Templates behandelt. Diese Templates sind Platzhalter für Metadaten. Zur Laufzeit werden diese Platzhalter durch konkrete Metadaten, aus externen Quellen, ersetzt [1].

In anderen Plug-in-Systemen wurde das Konzept der Generizität noch nicht umgesetzt [1]. Plug-in Frameworks wie *OSGi* und *Eclipse* setzen rein auf programmatische Kompositionen, wobei Entwickler die Komponenten explizit und manuel verbinden. Eine Verwendung für generische Metadaten haben sie nicht, da es keine automatischen Kompositionen gibt, bei deren die selbe Komponente mit unterschiedlichen Metadaten an verschiedenen Stellen im Quellcode benutzt werden würde. In Plux.NET werden automatische Kompositionen über Metadaten gesteuert. Da die Metadaten darüber entscheiden, welche Komponenten zusammengeführt werden, müssen sie sich bei jeder Wiederverwendung entsprechend unterscheiden. Generizität löst dieses Problem, indem die jeweiligen Metadaten von wiederverwendbaren Komponenten kontextbezogen angepasst werden.

OSGi ist eine hardwareunabhängige Softwareplattform und erleichtert es Anwendungen und ihre Dienste per Komponentenmodell (*Bundle/Service*) zu modularisieren und zu verwalten. Dabei steht die Komponente (*Bundle*) im Vordergrund, die ihre Schnittstelle (*Service*) per globaler Komponenten-Registry (*service registry*) als Contributor veröffentlicht. Hosts suchen in dieser Komponenten-Registry nach Services. Wenn ein Host mit einem Contributor verbunden wird, kann dieser nicht bestimmen, welcher Service (also welche Schnittstelle) vom Contributor genutzt wird. Wird die Kontrolle darüber benötigt, muss der Contributor eine eigene, konfigurierbare Schnittstelle zur Verfügung stellen [1].

Eclipse ist ein Java-basiertes System und benutzt XML um Extensions zu beschreiben. In Eclipse werden Extensions ebenfalls in einer globalen registry gehalten [1]. Referenziert werden sie dann über eine XML-Konfigurationsdatei. Die Metadaten in diesen Konfigurationsdateien geben an, welche *extension points* (z.B. Slots) die Extension beisteuert. Mit Hilfe von mehreren XML-Dateien, mit unterschiedlich ausgeprägten Metadaten, lassen sich mehrere Instanzen einer Extensions generieren. Hat die generierte Extension selbst extensions points, generiert Eclipse deren Namen nicht aus den XML-Metadaten. Stattdessen werden die Namen der extensions points in der Extension hardcodiert. In Plux sind die Namen der Slots und Plugs Teil der Metadaten und können für generische Extensions (Templates) angepasst werden.

Der entscheidende Vorteil von Plux.NET gegenüber den vorgestellten Systemen ist das Kompositionsmodell, welches automatische Kompositionen erlaubt

und Generizität und einfache Wiederverwendbarkeit von Komponenten überhaupt erst ermöglicht [1]. Dadurch wird der Programmieraufwand für Plug-in Entwickler reduziert und der Benutzer bekommt das Gefühl, Softwareteile ohne Programmier- oder Konfigurationsaufwand im Plug & Play-Modus zusammenstecken zu können. Zudem ist das Spezifizieren aller relevanten Informationen direkt im Sourcecode weniger fehleranfällig und einfacher zu warten, als beispielsweise die Deklaration in externen XML-Dateien. Weiterhin ist das Extrahieren von Metadaten und die Suche nach Extension in Plux.NET Plug-in basiert. Dies ermöglicht es, im Gegensatz zu den anderen Systemen, Extensions beispielsweise aus dem Netzwerk zu beziehen.

References

1. Wolfinger, R., Löberbauer, M., Jahn, M., Mössenböck, H.: Adding genericity to a plug-in framework. SIGPLAN Not. **46**(2) (October 2010) 93–102
2. Marquardt, K.: Patterns for plug-ins. Manolescu, D., Voelter, M., and Noble, J. Pattern Languages of Program Design **5** (2006) 301–317
3. Birsan, D.: On plug-ins and extensible architectures. Queue **3**(2) (March 2005) 40–46
4. Wolfinger, R., Dhungana, D., Prähofer, H., Mössenböck, H.: A component plug-in architecture for the .net platform. In: Proceedings of the 7th joint conference on Modular Programming Languages. JMLC'06, Berlin, Heidelberg, Springer-Verlag (2006) 287–305
5. Corporation, M.: Microsoft C# language specifications. Developer Series. Microsoft Press (2001)
6. Gruhn, V., Thiel, A.: Komponentenmodelle. DCOM, Javabeans, Enterprise Java Beans, CORBA. Addison-Wesley (2000)