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/

1 Plug-In's in der Softwareentwicklung

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 [Marquardt, 2006].

Der Grund für eine Plug-In-basierte Entwicklung ist Programme flexibler zu gestalten. Anwendungen sind oft monolithisch und schwergewichtig. Das kann bedeuten, das kleine Änderungen es erfordern, dass die ganze Software neu ausgeliefert werden muss. Dazu kommt, das ein Benutzer meist nur ein Bruchteil 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. Abbildung 1 vedeutlicht das anhand einer Gegenüberstellung. Das Zusammensetzten unterschiedlicher Programmteile ermöglicht es, Anwendungen in Hinsicht auf Größe, Komplexität und Kosten besser zu skalieren. Zudem erhalten die Nutzer, aber auch Drittentwickler, die Möglichkeit Programme um neue Funktionalitäten zu erweitern und an ihre Bedürfnisse anzupassen.

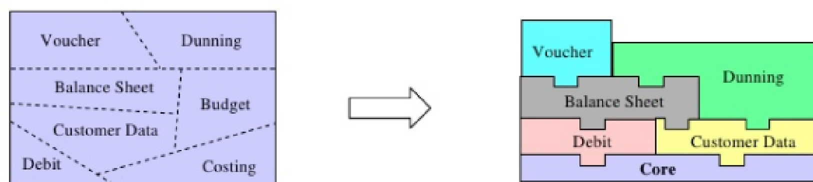

Fig. 1. Software-Monolith gegenüber einem schlanken Kern plus Plug-ins

Ein bekannter Vertreter Plug-in basierter Plattformen ist Eclipse, eine in Java geschriebene IDE.

Zusammenfassend lassen sich folgende Ziele für ein Plug-in-basierten Ansatz definieren:

– 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
– Keine XML-Konfigurationsdateien
– Sicherheitsrichtlinien (Wer kann ein Plug-in hinzufügen? Was darf ein Plug-in?)

In dieser Arbeit wird eine Plug-in Architektur vorgestellt, welche diese Zielsetzungen erfüllt. Dabei beruht die Architektur bzw. das Framework auf .NET Konzepten wie Attributen und Metadaten, um relevante Informationen zu einem Plug-in direkt im Sourcecode einer Anwendung zu spezifizieren. In [Wolfinger et al., 2006] 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 folglich auch für jede Methode des Interface eine Implementierung bereitstellen. Somit weiß jede Anwendung, die das Interface ebenfalls kennt, welche Methoden unumgänglich durch ein Plug-in implementiert wurden und welche Methoden sie aufrufen kann.

Ein essentielles Prinzip einer jeden Plug-in Architektur ist das *Plugging* [Wolfinger et al., 2006]. 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 nach Aussen geben. Dadurch ist definiert, welche Komponenten zusammengefügt - *geplugged* - werden können.

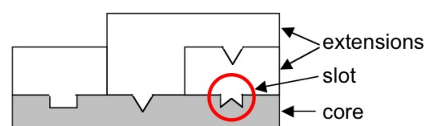

Fig. 2. Eine Extension (Plug-in) plugged in den Slot der Anwendung (core)

Wie Abbildung 2 vedeutlicht, können eine Reihe von Erweiterungen (*Extensions*) in die für sie vorgesehenen *Slots* zur Laufzeit geplugged werden. Ein Slot definiert dabei ein Interface und eine Liste von Parametern mit Namen und Typen. Eine Extension liefert dann 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 auch als kleine Anwendungen gesehen werden, welche eine Host-Anwendung um neue Dienste erweitert. Nichttriviale Dienste können dabei 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* [Wolfinger et al., 2006].

Unter dem Begiff *Discovery* versteht man das automatische Erkennen und Aktivieren von Plug-in's zur Lade- und Laufzeit einer Applikation [Wolfinger et al., 2006]. 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 [Wolfinger et al., 2010]. Generische Programmierung erlaubt es den Entwicklern zunächst 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 jeden 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. Anders formuliert gibt es in Java überhaupt keine generierten Typen. Stattdessen benutzt Java (*JRE*) zur Laufzeit *Object*-Referenzen und überlässt die Typüberprüfung allein dem Java-Compiler [Wolfinger et al., 2010].

Generizität wird in einem Plug-in System notwendig, wenn allgemein gehaltene Komponenten (dann als Erweiterungen), wie etwa Grid's, die in unterschiedlichen View's, unterschiedliche Datensätze anzeigen sollen, für diese Fälle wiederverwendet werden 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 einige technische Grundlagen zur Verfügung, auf dem der im nächsten Kapitel beschriebene Plug-in Ansatz basiert. An dieser

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

2.1 Reflexion

Reflexion ermöglicht es, die Struktur eines Programmes zur Laufzeit abzufragen und wenn nötig auch zu modifizieren. So ist es möglich während der Laufzeit Informationen über Klassen bzw. deren Instanzen abzufragen. Bei Methoden handelt es sich dabei beispielsweise um Information über die Sichtbarkeit oder die Datentypen von Übergabeparametern. Für die Realisierung von Reflexion ist das Speichern von Metainformationen[1] im Binärcode des Programms notwendig. .NET unterstützt Reflexion dahingehend, das 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;
}
```

Fig. 3. Liefern des Rückgabewertes einer Methode von einem gegeben Objekt

Abbildung 4 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 können. Sie sind damit das Pedant 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.

Abbildung 4 zeigt, wie der Klasse *StockTicker* das *[WebMethod]*-Attribut hinzugefügt wurde. Dieses Attribut zeigt an, das 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.

Die in dieser Ausarbeitung behandelte Plug-in Architektur benutzt Attribute um Informationen zu Plug-in-Komponenten zu deklarieren.

---

[1] Enthalten Information über Merkmale anderer Daten

[2] Common Runtime Language; Laufzeitumgebung von .NET

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

Fig. 4. Einer Klasse wird ein Attribut zugewiesen

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. 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.

Assembly können signiert werden. Dies wird häufig auch als das Vergeben eines starken Namens bezeichnet. Mit Hilfe dieses starken Namens und einer Versionsinformation eines Assemblies können später Komponenten identifiziert werden. Ein entsprechendes Beispiel einer deklarativen Versionierung zeigt Abbildung 5. Die so durch ein Attribut markierte Version lässt sich später per Reflexion auslesen.

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

Fig. 5. Einer Klasse wird ein Attribut zugewiesen

In der vorgestellten Plug-in Architektur 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 nicht nur Code sondern auch Metadaten, welche die Symbolinformationen aller Typen, Methoden und Felder in diesem beschreiben. Die Metadaten werden automatisch vom Compiler aus dem Sourcecode erzeugt. Auch hier ermöglicht es .NET die Metadaten via Reflexion zur Laufzeit aus dem Assembly auszulesen. Der Code in Abbildung 6 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
  }
}
```

Fig. 6. Reflektion erlaubt das Auslesen von Metadaten aus Assemblies zur Laufzeit

Die in dieser Ausarbeitung erörterte Plug-in Architektur 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 [Wolfinger et al., 2010] stellen die Autoren eine Plug-in Architektur bzw. ein Plug-in Framework für Microsoft .NET vor. Dieses Plug-in Framework ermöglicht Software-Kompositionen durch eine Art *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 [Gruhn and Thiel, 2000] einen Rahmen für die Entwicklung und Ausführung von Komponenten fest. Dabei spielen strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten eine wichtige Rolle.

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 7 verdeutlicht. Man spricht auch von *host extensions* bzw. *Host* oder *contributor extensions* bzw. *Contributor*, jenachdem ob eine Extensionen einen Slot öffnet oder einen Plug bereitstellt. Das Zusammenfügen von einem Plug in ein Slot nennt man, wie einleitend erwähnt, Plugging.

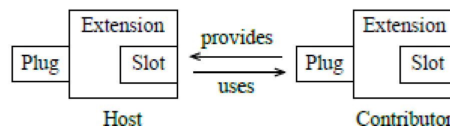

Fig. 7. Slot-Plug-Beziehung von zwei Extensions

Konkret spezifiziert ein Slot 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 auch hier 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 einen eindeutigen Namen identifiziert. Dies geschieht ebenfalls über ein Metaelement (*[Slotdefinition("name")]* und *[Plug("name")]*). Ein Plug passt zu einem Slot, wenn ihre Namen übereinstimmen.

Abbildung 8 verdeutlicht dies anhand von Sourcecode. Es wird ein Host bzw. Slot mit dem Namen *Logger* definiert, welcher Lognachrichten mit einem Timestamp ausgeben soll. 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. Zusätzlich wird ein konkretes Zeitformat (*hh:mm:ss*) über den Parameterwert festgelegt.

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

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

Fig. 8. Slot-Definition (links) und passender Plug (rechts)

Das Plux.NET Framework selbst ist Plug-in basiert. Das bedeutet, das die eigene 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 Abbildung 9). Die Anwendung hat weiterhin einen Slot *Logger*, welcher bereits wie in Abbildung 8 gezeigt, implementiert wurde und die eigentliche Funktionalität bereitstellt.

Der Anwendung wird im Konstruktor beim Laden durch das Framework ein Objekt mit den Metadaten der assozierten 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, ebenfalls definiert in Abbildung 8. Dies ist jedoch dynamisch, sodass 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.

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

```
[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 */
 }
}
```

Fig. 9. Applikation mit Plug zum Framework und Slot zum Logger

Der Slot bzw. das Interface *ILogger*, die Plugs bzw. die Klassen *ConsoleLogger* (Abbildung 8) und *MyApp* (Abbildung 9) werden zu DLL's (Assemblys) 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 Kompositionsmodel, 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, passenden 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 Metainformationen sowie die Suche nach den Extensions nicht integraler Bestandteil des Frameworks, sondern ist ebenfalls 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

Bedingt dadurch, das die Metadaten der Slots und Plugs definieren, welche Host- bzw. Contributor-Extensions zusammen passen, ist es nicht so einfach möglich allgemein gehaltene Extensions bzw. Komponenten wiederzuverwenden. Man müsste jedesmal neue Metadaten definieren und diese verschieden ausprägen.

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 Roa	Armc
3	Microsoft C	(800) 426-9900	1 Microsoft Way	Redn

search for: search in: *Name & Addres* search mode: *Beginning of field*

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: *Description* search mode: *Anywhere in field*

Fig. 10. Artikel- und Kunden-View als beispielhafte UI

Abbildung 10 zeigt eine Applikation mit zwei Views. Die eine View zeigt in einem Grid Kundendaten an und eine weitere View zeigt Artikeldaten in einem anderen Grid an. Weiterhin hat jede View ein Filter-Panel mit den die Datensätze sortiert bzw. durchsucht werden können. Eine entsprechende Extension für den View (Host) würde einen Slot öffnen, an den zunächst beliebige 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. Beide benötigen, um Daten anzeigen und sortieren zu können wiederum einen Slot für die eigentliche Datenquelle. Schematisch wird dies in Abbildung 12 gezeigt. Abbildung 11 verdeutlicht dies für das Grid anhand von Sourcecode.

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 auch 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 Abbildung 11 zu sehen, hat die *Grid*-Extension ein *Control*-Plug für die View und ein *DataSource*-Slot für die Datenbereitstellung. Die Problematik die hierbei entsteht ist, das wenn das Framework die *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 also in das Grid für den Kunden-View als auch in das für den Artikel-View. Das betrifft gleichermaßen die Filter-Extension. Dies wird

```
[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;
}
```

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

Fig. 11. Slot-Definition für das Control

auch durch das linke Schema in Abbildung 12 verdeutlicht. Eine weitere Problematik dabei ist, dass das Framework jedes *Control*-Plug eines Filter oder Grids in jeden View plugged. Man kann also nicht spezifizieren, 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. Die rechte Komposition in Abbildung 12 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.

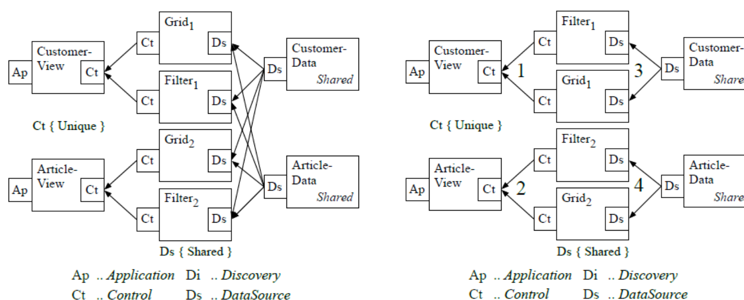

Fig. 12. Inkorrekte (links) und korrekte (rechts) Komposition

Zusammenfassend müsste man, für eine korrekte Komposition eines Artikel- und Kunden-Views, die Controls mit den Views und die Datenquellen mit den Controls selektiv verbinden. Ohne einen generischen Ansatz stehen zwei Lösungswege zur Verfügung. Zum einen kann der Komposer deaktiviert werden und die Extensions werden programmatisch verbunden. Jedoch widerpricht das genau 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, da die View keine Controls integrieren kann, welche nicht zur Übersetzungszeit bekannt waren. Hinzu kommt, das wenn alle Datenquellen den selben Plug-Name nutzen, das andere Hosts dann ebenfalls manuell zusammengesetzt werden müssten, wenn diese auch die Datenquelle benutzen wollen.

Eine andere Möglichkeit, um eine korrekte Komposition eines Artikel- und Kunden-Views zu erreichen, wäre es, wie bereits eingangs erwähnt, verschiedene Grid- und Filterextensions mit unterschiedlichen Metadaten zu schreiben, sodass sie für ihren jeweiligen Kontext (Artikel- oder Kunden-View) passend sind. Es werden also für jede Wiederverwendung neue Extensions (Subklassen) erzeugt (siehe Abbildung 13). Beispielsweise könnte dann eine *ArtikelGrid*-Klasse von der bereits bekannten *Grid*-Klasse erben und dort dann einen konkret ausgeprägenten 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 { ... }
```

Fig. 13. Wiederverwendung von Extensions über sub-classing

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

Für einen generischen Ansatz sollte es also möglich sein, Extensions mit Metadaten zu generieren, welche unabhängig von der implementierenden Klasse sind. Weiterhin soll der generische Ansatz es ermöglichen, mehrere Extensions mit unterschiedlichen Metadaten aus einer einzigen Klasse zu generieren.

Ein generisches Plug-in in Plux.NET beinhaltet ein oder mehrere *extension templates*. Ein extension template wiederum 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, nocheinmal unter dem Gesichtspunkt des generischen Ansatzes betrachtet (siehe dazu Abbildung 14).

Das Template *GridTemplate* hat vier Platzhalter. Den $< Grid >$-Platzhalter für den Extension-Name, den $< Control >$-Platzhalter für den Plug-Namen,

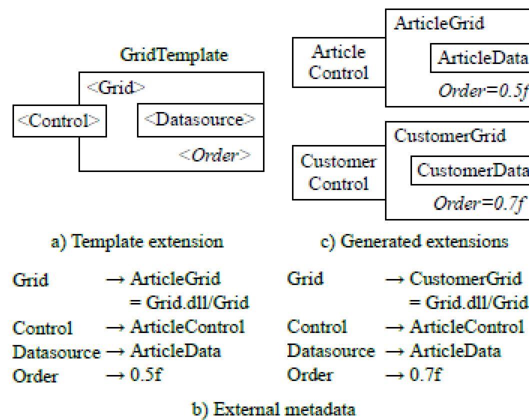

Fig. 14. Grid extensions generiert aus einem Template und externen Metadaten

den $< DataSource >$-Platzhalter für den Slot-Name und den $< Order >$-Platzhalter für ein Parameterwert. Die externen Metadaten (b)) spezifizieren, das zwei Extensions aus dem Template generiert werden sollen. 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.

Der linke Teil in der Abbildung 15 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 weiterhin 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 Slotdefinition auf welchen ein Slot oder Plug (namentlich) basiert. Beispielsweise basiert der Template-Slot $< DataSource >$ auf der Slot-Definition *DataSource* wie in Abbildung 11 zu sehen.

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

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

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

Fig. 15. Grid extensions generiert aus einem Template und externen Metadaten

Der rechte Teil der Abbildung 15 zeigt die finale *Grid*-Klasse als Resultat aus dem Template (links) und den externen Metadaten (mitte).

Wurden bisher Slots über ihre Namen, aus dem durch das Framework im Konstruktor übergebene Metadaten-Objekt der assozierten Extensions extrahiert und referenziert (vgl. Abbildung 9), kommen für den template-basierten Ansatz Indizies zum Tragen. 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 Abbildung 16.

```
[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];
  }
  ...
}
```

Fig. 16. Slot-Referenzierung über Indizes statt Namen

Das Generieren von Extensions aus Templates bedeutet im Kern 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 Abbildung 15), untersucht ein spezieller *template analyzer* das Assembly nach dem *[Template]*-Attribut. Wird dies gefunden, sucht der template analyzer dann nach einer Konfigurationsdatei mit Metadaten, wobei der Template-Name und der Plug-in Assembly-Name übereinstimmen müssen (siehe Metadaten in Abbildung 15; *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 [Wolfinger et al., 2010] 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.

In anderen Plug-in-Systemen wurde das Konzept der Generizität noch nicht umgesetzt [Wolfinger et al., 2010]. Plug-in Frameworks wie *OSGi* und *Eclipse* setzen rein auf programmatische Kompositionen, wobei Entwickler die Komponenten explizit und manuel verbinden. Diese Frameworks können natürlich generische Klassen nutzen, eine Verwendung für generische Metadaten haben sie allerdings 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 Komponten, je nach Kontext, 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 wiederum 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.

Eclipse ist ein Java-basiertes System und benutzt XML um Extensions zu beschreiben. In Eclipse werden Extensions ebenfalls in einer globalen registry gehalten [Wolfinger et al., 2010]. 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 jedoch 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. Dadurch wird der Programmieraufwand für Plug-in-Entwickler reduziert und der Benutzer bekommt das Gefühl, Softwareteile ohne Programmier- oder Konfigurationsaufwand im Plug&Play-Modus zusam-

menstecken 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 auch aus dem Netzwerk zu beziehen.

References

[Gruhn and Thiel, 2000] Gruhn, V. and Thiel, A. (2000). *Komponentenmodelle. DCOM, Javabeans, Enterprise Java Beans, CORBA*. Addison-Wesley.

[Marquardt, 2006] Marquardt, K. (2006). Patterns for plug-ins. *Manolescu, D., Voelter, M., and Noble, J. Pattern Languages of Program Design*, 5:301–317.

[Wolfinger et al., 2006] Wolfinger, R., Dhungana, D., Prähofer, H., and Mössenböck, H. (2006). A component plug-in architecture for the .net platform. In *Proceedings of the 7th joint conference on Modular Programming Languages*, JMLC'06, pages 287–305, Berlin, Heidelberg. Springer-Verlag.

[Wolfinger et al., 2010] Wolfinger, R., Löberbauer, M., Jahn, M., and Mössenböck, H. (2010). Adding genericity to a plug-in framework. *SIGPLAN Not.*, 46(2):93–102.