Ralf Kruthoff-Brüwer

(E-Mail: r.kruthoff-bruewer@fh-osnabrueck.de)
schloss 2009 das Studium der Medieninformatik ab und war anschließend in einem 6-monatigen Projekt der Science to Business GmbH beschäftigt, in dem es um den Entwurf eines skalierbaren Serversystems zur Auslieferung von Daten an PVR-Systeme ging. Seit September 2009 arbeitet er im Projekt "Strategische Flexibilität durch komponentenbasierte Software-Entwicklung" an Modellen zur komponentenbasierten Softwareentwicklung.

Prof. Dr. Frank M. Thiesing

(E-Mail: f.thiesing@fhos.de)
ist Professor für Software-Engineering und Mathematik in der Fakultät Ingenieurwissenschaften und Informatik der Stiftung Fachhochschule Osnabrück. Sein Forschungsschwerpunkt ist die komponentenbasierte Softwareentwicklung.

Frank Nordemann

(E-Mail: info@franknordemann.de)
hat sein Studium der Medieninformatik an der Fachhochschule Osnabrück im Sommersemester 2008 abgeschlossen. In seiner Diplomarbeit hat er sich mit der Entwicklung eines leichtgewichtigen Komponentenmodells befasst und in diesem Zusammenhang Methoden zur Komponentenkomposition, -aggregation und -introspektion entwickelt. Mit seiner Tätigkeit in einem Forschungsprojekt führt er Untersuchungen im Bereich der komponentenbasierten Softwareentwicklung fort.

Flexibilisierung von Komponentenarchitekturen durch das Managed Extensibility Framework

Ein Teil des Anfang 2010 erscheinenden .NET-Frameworks 4.0 wird das Managed Extensibility Framework (MEF) sein. MEF dient zur einfachen Erweiterung von Anwendungen und ermöglicht einen reibungslosen Austausch einzelner Komponenten. Von Haus aus bietet MEF ein Programmiermodell, welches mit Attributen arbeitet, um angebotene und benötigte Schnittstellen einer Komponente direkt im Quellcode zu definieren. Diese werden anschließend von MEF mittels Reflektion zur Auflösung von Komponentenabhängigkeiten ausgelesen und verarbeitet. MEF bietet als integraler Bestandteil des .NET-Framework 4.0 also eine Möglichkeit zur standardisierten Beschreibung von Komponenten und deren Schnittstellen. So können Komponenten über Projekt- und unter Umständen sogar Firmengrenzen hinweg wiederverwendbar gestaltet werden.

Einleitung

In vielen laufenden Projekten werden bereits Frameworks eingesetzt, die eine ähnliche Funktionalität bieten wie das Managed Extensibility Framework. Sehr verbreitet in diesem Segment sind Frameworks wie Spring [spr], Castle [cas] oder Unity [uni], die allesamt auf dem Dependency Injection (DI) Prinzip, einer Spezialisierung des Inversion of Control (IoC) Musters [Fow04], basieren. Die dort verwendeten Prinzipien haben sich in der modernen Softwareentwicklung bewährt. Abhängig von den Projektanforderungen haben sich verschiedene DI-Container etabliert.

Dieses sehr heterogene Feld von DI-Containern senkt das Maß an Wiederverwendbarkeit einzelner Komponenten enorm. Eine Komponente kann nur innerhalb des speziellen Modells wiederverwendet werden. Ein projektübergreifendes oder sogar globales Komponenten-Repository ist durch diesen Umstand kaum zu realisieren. Einzelne Projektteams möchten jedoch nicht auf die Vorzüge ihres speziellen Modells verzichten. MEF bietet durch ein sehr anpassbares Programmiermodell einen guten Ansatz zur Flexibilisierung von komponentenbasierten Ansätzen. So können die Komponenten eines Modells in anderen verwendbar gemacht werden.

Dieser Artikel soll eine Einführung in das Managed Extensibility Framework und das vorhandene Attributed Programming Model liefern. Anschließend wird gezeigt, wie ein eigenes Programmiermodell erstellt und genutzt werden kann, um MEF in Verbindung mit einem eigenen Komponentenmodell (COMPASS) nutzen zu können.

Überblick über das Managed Extensibility Framework

Das MEF teilt sich in drei Schichten (vgl. Abbildung 1). Der Kern, die Container-Schicht, bietet Funktionalitäten zur Auflösung von Abhängigkeiten einzelner Komponenten und ist somit für die Komposition von Komponenten zuständig. Die Composition Primitives kapseln die konkreten Programmiermodelle vom Kern ab. Somit bedarf es bei der Erstellung eines eigenen Programmiermodells keinerlei Anpassung in der Container-Schicht. Das Attributed Programming Model wird bereits mit MEF ausgeliefert. Parallel zu diesem kann ein eigenes, hier als Custom

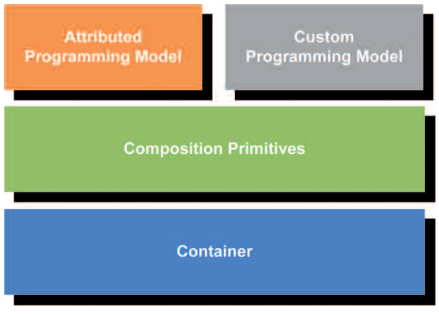

Abb. 1: Architektur des Managed Extensibility Framework

Programming Model bezeichnetes, Programmiermodell erstellt werden.

Imports und Exports

Um Abhängigkeiten unter Komponenten in MEF auszudrücken, werden sogenannte *Imports* und *Exports* verwendet. Ein Import definiert eine benötigte Funktionalität einer anderen Komponente. Ein Export definiert eine Schnittstelle die diese Abhängigkeit erfüllen kann. Schnittstellen werden MEF-intern durch einen Vertragsnamen und selbstdefinierte Metadaten ausgedrückt.

Das Attributed Programming Model

Das bereits mit dem MEF ausgelieferte Attributed Programming Model bietet einem Entwickler die Möglichkeit, die Definition von benötigten und angebotenen Schnittstellen direkt im Quellcode vorzunehmen. Somit kann man auf sehr einfache Weise Komponenten definieren.

Code-Ausschnitt 1 zeigt die Definition einer angebotenen Schnittstelle durch das Export-Attribut. Als Parameter wird der Typ der Schnittstelle als Vertragsname verwendet. Hier kann auch eine einfache Zeichenkette (String) eingesetzt werden, um die Bezeichnung der Schnittstellen unabhängig vom echten C#-Interface zu definieren.

```
[Export(typeof(IMovieLister))]
public class XML_MovieLister {...}
```

Code-Ausschnitt 1: Export-Definition nach dem Attributed Programming Model

Um diese durch das Export-Attribut definierten Schnittstellen zu nutzen wird das Import-Attribut verwendet (siehe Code-Ausschnitt 2). Auch hier gibt es weitere Möglichkeiten den Vertragsnamen anzugeben.

```
[Import]
public IMovieLister lister { get; set; }
```

Code-Ausschnitt 2: Import-Definition nach dem Attributed Programming Model

Die Composition Primitives

Basis des Attributed Programming Model und auch des zu erstellenden eigenen Programmiermodells sind die Composition Primitives (siehe Fehler! Verweisquelle konnte nicht gefunden werden.). Sie kapseln das Programmiermodell von den MEF-Kernklassen. Zur Erstellung eines eigenen Programmiermodells müssen die Composition Primitives zumindest teilweise implementiert oder erweitert werden.

Im MEF wird jede Instanz einer Komponente durch einen ComposablePart repräsentiert. Er enthält z. B. Informationen zu den benötigten und angebotenen Schnittstellen (ImportDefinition, Export Definition).

Vor der Instanziierung wird eine Komponenten durch eine ComposablePart Definition beschrieben. Eine ComposablePartDefinition stellt eine Fabrik für einen speziellen ComposablePart bereit. Durch die Methode CreatePart wird eine konkrete Instanz der Komponente erzeugt. Zwischen einer ComposablePartDefinition und einem ComposablePart steht in gewisser Weise eine Art Typ-Instanz-Beziehung.

In den beiden Primitiven ComposablePart und ComposablePartDefinition werden zur Beschreibung der angebotenen und benötigten Schnittstellen die Klassen ImportDefinition und ExportDefinition benutzt. Eine ExportDefinition besteht aus dem Feld ContractName (String), das den Vertragsnamen enthält, und einer Liste von Metadaten (Dictionary). Um nun entscheiden zu können, ob eine ExportDefintion die Anforderungen einer ImportDefinition erfüllt, wird in der ImportDefinition ein LINQ-Ausdruck (Constraint) [msd] definiert. Code-Ausschnitt 3 zeigt beispielhaft einen solchen Ausdruck. Hier wird ähnlich wie bei einer Datenbankabfrage eine Export Definition selektiert, die im Feld Contract Name den Wert „IMovieLister“ enthält.

```
(ExportDefinition) def =>
        def.ContractName == "IMovieLister";
```

Code-Ausschnitt 3: LINQ-Ausdruck für Import-Constraint

Für den Kompositionsvorgang müssen nun alle ComposablePartDefinitions in den MEF-Container eingefügt werden. Dies kann manuell geschehen oder alternativ automatisiert über einen sogenannten Katalog, mit dem sozusagen direkt mehrere ComposablePartDefinitions adressiert werden. Ein ComposablePartCatalog beinhaltet also Auswahl von ComposablePart Definitions. Wie diese Auswahl getroffen wird, hängt von der konkreten Implementierung des Katalogs ab. Im Attributed Programming Model werden z. B. Kataloge zum Sammeln aller Parts in einem Assembly oder in einem Verzeichnis angeboten.

Das COMPASS-Komponentenmodell

Das im Rahmen eines Forschungsprojekts der Fachhochschule Osnabrück in Zusam-

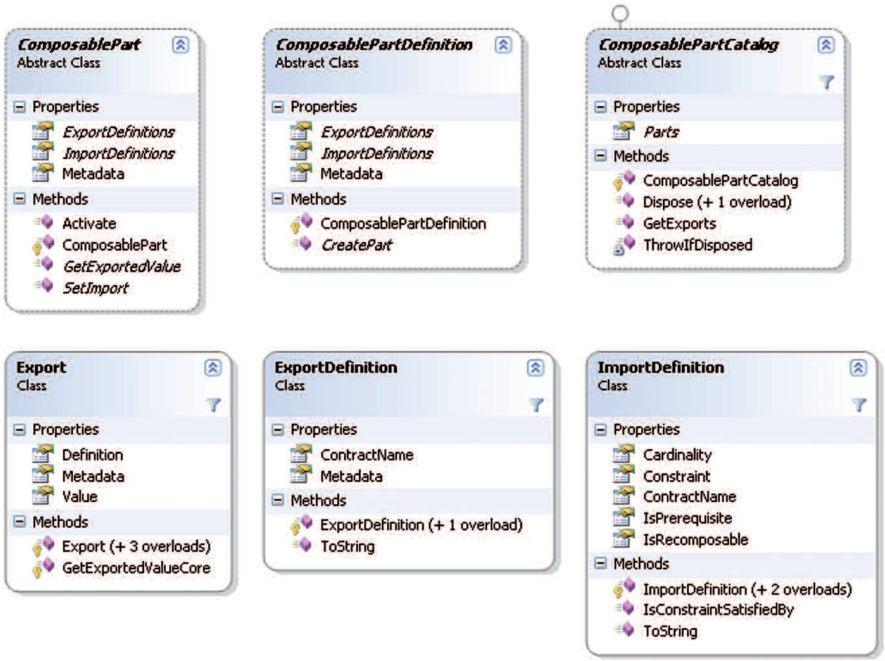

Abb. 2: Composition Primitives

menarbeit mit der ROSEN Technology and Research Center GmbH entwickelte Komponentenmodell COMPASS (COM Ponent based and Architecture centric development of Software Systems) verwendet in umfangreicher Form das Spring .NET-Framework, um Komponenten zu bilden und deren Abhängigkeiten aufzulösen. Es zeichnet sich besonders durch die Leichtgewichtigkeit des Programmiermodells aus. Entwicklern ist mithilfe des COMPASS-Modells ein sehr schneller Einstieg in die komponentenbasierte Softwareentwicklung möglich.

Komponenten und deren Abhängigkeiten werden durch XML-Strukturen beschrieben, die den Komponenten beigelegt werden. Eine beispielhafte Spring-Konfiguration ist in Code-Ausschnitt 4 gezeigt. Hier wird ein Programm erzeugt, das Filminformationen aus einer XML-Datei lädt und sie in einer grafischen Oberfläche anzeigt. Das Programm besteht aus zwei Komponenten. Dem XML MovieLister, der das Lesen aus der Datei übernimmt, und der MovieGUI, die für die Anzeige der Daten zuständig ist. Über das Tag property wird der Oberfläche eine Referenz auf den XMLMovieLister übergeben. Bei Ausführung des Programms ist eine Instanz des XMLMovieLister nun über die Member-Variable Lister der MovieGUI verfügbar.

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net">
  <object id="MovieGUI " type="Test.MovieGUI, Example ">
    <property name="Lister" ref="XMLMovieLister"/>
  </object>
  <object id="XMLMovieLister"
            type="Example.XMLMovieLister, Example"/>
</objects>
```

Code-Ausschnitt 4:
Beispiel einer Spring-Konfiguration

Ziel der Kombination mit MEF ist es, in der Konfiguration einer COMPASS-basierten Software nun Schnittstellen definieren zu können, die MEF zur Komposition nutzen kann. Es sollen also MEF-Imports und -Exports in der Spring-Konfiguration abgebildet und bei Ausführung verarbeitet werden.

Kombination der Programmiermodelle

Die Kombination der Programmiermodelle beinhaltet Anpassungen in beiden Modellen. Zum einen muss ein eigenes MEF-Programmiermodell mithilfe der Composition Primitives erstellt werden und zum anderen muss das COMPASS-Modell so angepasst werden, dass eine Definition von Schnittstellen zu anderen MEF-Komponenten möglich wird. Ziel ist es, Schnittstellen zu definieren, die nicht nur innerhalb des COMPASS-Modells gültig sind.

Anpassungen im COMPASS Modell

Um keine Eingriffe in die internen Strukturen des Spring.NET-Frameworks und dessen XML-Parser vornehmen zu müssen, ist die Erstellung zweier Hilfsklassen ratsam, die zum einen das zu exportierende oder zu importierende Objekt (Typ Object) enthalten und zum anderen ein Feld, in dem der Vertragsname hinterlegt ist (Typ String). Die Objektinstanziierung wird von Spring übernommen. Dabei kann mithilfe des Interface IObjectPostProcessors [spf] Einfluss genommen werden. Hier kann überprüft werden, ob es sich bei dem zu erstellenden Objekt um eine der Hilfsklassen handelt. Ist dies der Fall, wird das in der Hilfsklasse gekapselte Objekt zusammen mit dem Vertragsnamen in eine Liste eingefügt. Für Imports und Exports werden zwei separate Listen erstellt. Sind alle Objekte erstellt, können die beiden Listen, die alle für MEF gültigen Imports und Exports enthalten, an das MEF-Programmiermodell gereicht werden oder dort später durch einen Katalog verwaltet werden.

```
...
<object id="XML_MovieLister"
            type="Example.XML_MovieLister, Example">
...
<object id="MEF_Export "
            type="MEF_Helper.MEF_Export, Example">
  <property name="ExportedObject"
            ref="XML_MovieLister"/>
  <property name="ContractName"
            value="XML_MovieLister"/>
</object>
...
```

Code-Ausschnitt 5: MEF-Export in COMPASS-Konfiguration definieren

Der Code-Ausschnitt 5 zeigt am Beispiel der Konfiguration einer COMPASS-Komponente, wie dieser ein in MEF verwendbarer Export hinzugefügt werden kann. Die so erstellte Komponente könnte nun von anderen Komponenten importiert werden. Dabei spielt es keine Rolle, ob dieser Import durch ein Attribut, über eine COMPASS-Konfigurationsdatei oder ein völlig anderes Programmiermodell definiert ist.

Entwurf des MEF Programmiermodells

Zu erstellen ist die in Fehler! Verweisquelle konnte nicht gefunden werden. gezeigte Struktur. Die abstrakten Primitiven ComposablePartCatalog, ComposablePart und ComposablePartDefinition müssen implementiert werden. Diese werden im Folgenden als COMPASS_Composable PartCatalog, COMPASS_ComposablePart und COMPASS_ComposablePartDefinition bezeichnet.

Um nun aus den COMPASS-Komponenten für MEF verwertbare COMPASS_ComposableParts zu erstellen, muss durch die Listen der Imports und Exports aus dem COMPASS-Container iteriert werden. Für jede Komponente, die einen MEF-Export oder –Import enthält, wird eine COMPASS_ComposablePartDefinition in den COMPASS_ComposablePartCatalog eingefügt. Diese werden vorher mit den in den Listen aufgeführten Imports und Exports versehen. Repräsentiert werden die Imports und Exports durch die Klassen ImportDefinition und ExportDefinition. Diese Klassen können ohne eigene Implementierung aus den Composition Primitives übernommen werden. Die auf diese Art und Weise erstellte Katalogstruktur kann genau wie die bereits in MEF vorhandenen Kataloge in den Kompositionsvorgang eingebunden werden. So ist es möglich, COMPASS-Komponenten mit MEF-Schnittstellen zu versehen. Diese sind nun kompatibel zu anderen MEF-Pro-

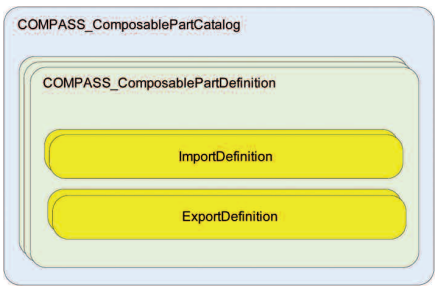

Abb. 3: Das zu erstellende Programmiermodell

grammiermodell (z. B. dem Attributed Programming Model) oder zu Komponentenmodellen mit MEF-Anbindung, wie z. B. dem Unity-Container.

Zusammenfassung

Alles in allem bietet das Managed Extensibility Framework eine solide Grundlage zur Entwicklung komponentenbasierter Software. Mit dem Attributed Programming Model ist ein sehr einfacher Einstieg in MEF möglich und die ersten Anwendungen lassen sich in kürzester Zeit entwickeln. Auch die Erstellung eines Plug-In-Mechanismus für bestehende Anwendungen ist mit dem Attributed Programming Model problemlos möglich.

Ist bereits ein komponentenbasierter Ansatz, z. B. durch ein externes Framework gegeben, kann durch die Erweiterung der Composition Primitives Funktionalität von MEF in die bestehende Architektur eingebracht werden. Durch eine Anbindung des eigenen an das MEF-Komponentenmodell wird also eine erhebliche Flexibilisierung des bestehenden Ansatzes erreicht. Diese äußert sich in einer gesteigerten Interoperabilität. Die Vorteile des im .NET-Framework integrierten MEF-Komponentenmodells können genutzt werden, ohne eigene Konzepte verwerfen zu müssen. ■

Quellen

[spr] http://www.springframework.net/

[cas] http://www.castleproject.org/

[uni] http://www.codeplex.com/unity/

[Fow04] Fowler, M. (2004, Januar 23). Inversion of Control Containers and the Dependency Injection pattern. Retrieved November 7, 2009, from http://martinfowler.com/articles/injection.html

[msd] http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

[spf] http://www.springframework.net/doc-latest/reference/html/objects.html#d4e1794