Boris Wehrle
(Boris.Wehrle@aitgmbh.de)
ist Senior Consultant bei der AIT GmbH & Co. KG. Er berät Unternehmen bei der Softwareentwicklung auf Basis von Microsoft-Technologien. Seine Schwerpunkte liegen in der Konzeption und Entwicklung von verteilten Anwendungen im industriellen Bereich.

Herausforderungen bei der Entwicklung individualisierter Standardsoftware – Wege aus dem Verwaltungschaos

Standard und individualisiert – schließt sich das nicht aus? Keineswegs! Um sich Wettbewerbsvorteile zu verschaffen, werden von Kunden individuelle Lösungen benötigt und gefordert. Gleichzeitig soll eine solche Lösung kostengünstig umgesetzt und fortlaufend weiterentwickelt werden. Wie ist dieser Spagat zwischen einer Individuallösung und einem Standardprodukt zu schaffen? Durch eine Architektur, die eine flexibel konfigurierbare und weitreichend erweiterbare Standardsoftware ermöglicht. Der Artikel zeigt, wie man eine solche Lösung aufbaut und wie man kundenspezifische Anpassungen und Erweiterungen im Lebenszyklus der Anwendung im Griff behält, OHNE im Verwaltungschaos zu enden.

Die Klassiker: Die Vorstufe des Chaos

In der Praxis trifft man häufig auf zwei Ansätze zur Abbildung von kundenindividuellen Lösungen, basierend auf Branches oder Konfigurationen. Beide Ansätze funktionieren bis zu einem bestimmten Punkt in der Regel sehr gut. Übersteigt die Individualisierung aber ein gewisses Maß oder gibt es sehr viele Kundenvarianten, landet man allerdings schnell im Chaos.

Der Aufwand zum Abgleich der verschiedenen Lösungen übersteigt schnell den eigentlichen Entwicklungsaufwand. Eine Anpassung des Vorgehens und der damit einhergehenden Architektur sind aufgrund der bis dahin bereits zahlreich ausgelieferten Lösungen nur noch sehr schwer möglich.

Das Gemeine an der Sache: Man merkt es erst, wenn es zu spät ist! Damit Sie einordnen können, ob Sie bereits in der Falle stecken und es dringend Zeit zum Handeln ist, werden die Klassiker kurz vorgestellt.

Individualisierung durch Branches

Die am häufigsten anzutreffende Variante ist eine Verwaltung von kundenindividuellen Lösungen mithilfe einer Branching-Struktur, bei der mehrere Zweige (Branches) des Quellcodes in der Versionsverwaltung voneinander getrennt bearbeitet werden. Auf dem Main-Branch bzw. den Development-Branches wird eine Standardlösung entwickelt. Basierend auf dieser werden kundenspezifische Branches, wie in Abbildung 1 dargestellt, abgezweigt. In diesen werden kundenspezifische Erweiterungen oder Anpassungen bestehender Komponenten eingepflegt.

Neue im Hauptzweig (Main Branch) bereitgestellte Funktionen können mithilfe eines Merge-Vorgangs in eine Kundenlösung übernommen werden. Dies ist mithilfe der Versionsverwaltung relativ einfach und sicher zu bewerkstelligen. Wie geht man aber vor, wenn man ein für den Kunden A entwickeltes Modul auch beim Kunden B integrierten möchte?

Bei Verwendung der Funktionen der Quellcodeverwaltung muss man beim Zu-

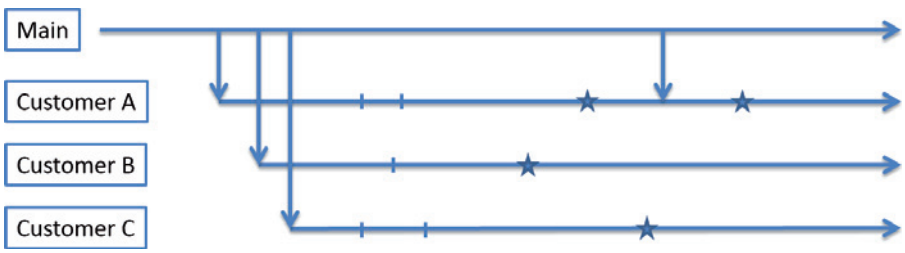

Abb. 1: Abbildung von Kundenindividualisierungen über eine Branching-Struktur

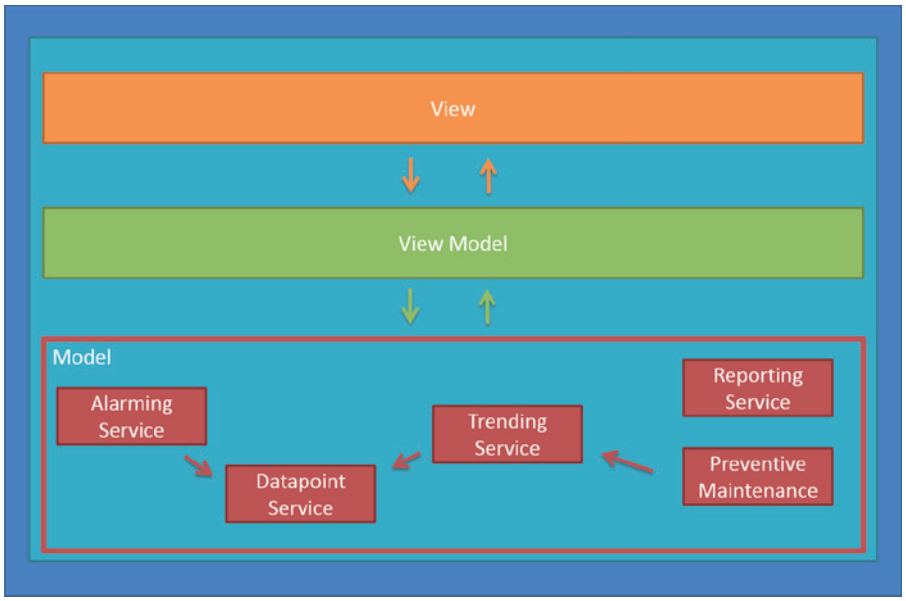

Abb. 2: Aufbau einer Anwendung nach dem MVVM-Pattern

sammenführen (Merge) zunächst einen Weg über den Hauptzweig einschlagen. Dies führt allerdings auch dazu, dass beim nächsten Merge die Funktion auch bei Kunde C landet, der diese überhaupt nicht benötigt.

Ein vergleichbares Verhalten zeigt sich, wenn man einen Bugfix, den man für den Kunden A bereitgestellt hat, auch allgemein verfügbar machen will. Dies macht einen selektiven Merge notwendig. Leicht wird hierbei auch Code übernommen, der für andere Kunden nicht bereitgestellt werden soll. Als Problemumgehung wird deshalb häufig ein Bugfix manuell durch Copy-and-paste in den Main Branch übernommen – ein ebenfalls sehr fehleranfälliges Verfahren.

Der Überblick darüber, bei welchem Kunden welche Funktion und welcher Bugfix bereits eingespielt wurde, geht schnell verloren. Sobald die Anzahl von Individuallösungen zehn übersteigt, landet man schnell im Chaos. Ein wesentlicher Vorteil dieser Lösung ist es jedoch, dass diese mit jeder Form von Architektur (also auch ohne) funktioniert.

Individualisierung durch Konfiguration

Ein weiterer Ansatz der häufig anzutreffen ist, basiert auf einer flexibel über eine Konfigurationsdatei anpassbare Standardsoftware. Alle Komponenten, auch für einzelne Kunden entwickelte Lösungen, werden auf einem Branch entwickelt. Die einzelnen Funktionen werden über eine Konfigurationsdatei bzw. eine Lizenzierungskomponente freigegeben und deren Verhalten gegebenenfalls hierdurch angepasst.

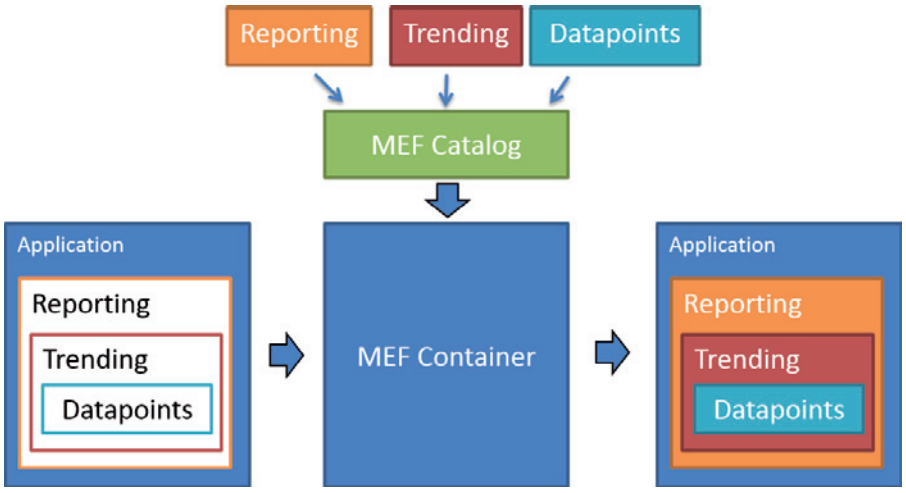

Abb. 3: Zusammenstellung von Komponenten zu einer Anwendung mithilfe des Managed Extensibility Frameworks (MEF)

Bugfixes können somit an zentraler Stelle vorgenommen werden. Fehleranfällige Merge-Vorgänge sind nicht notwendig. Die Lösung bietet sich insbesondere an, wenn die Kundenindividualisierungen im Umfang relativ gering sind. Bei starken Unterschieden in den Funktionsumfängen geht schnell der Überblick über die verschiedenen Konfigurationsmöglichkeiten verloren. Umfangreiche Tests vor einer Auslieferung an einen Kunden werden notwendig.

Der komponentenorientierte Ansatz: Der Ausweg aus dem Chaos

Die beiden Klassiker kommen bei einem hohen Grad an Individualisierung, wie er zum Beispiel im Maschinenbau sehr häufig vorkommt, an ihre Grenzen. Mit zunehmender Kundenanzahl wird die Verwaltung immer komplexer und verschlingt immer mehr wertvolle Ressourcen. Das Ganze ist ein schleichender Prozess, sodass man ihn anfangs gar nicht wahrnimmt, sondern erst, wenn es zu spät ist und nur noch mit hohem Aufwand eine Korrektur möglich ist.

Deshalb sollte man, wenn es sich abzeichnet, dass Kunden eine hohe Individualisierung benötigen, von vornherein einen anderen Weg einschlagen. Eine Berücksichtigung dieser Anforderung in der Architektur hilft spätere Kopfschmerzen zu vermeiden.

Zunächst gilt es, sich vom Gedanken zu lösen, dass man EINE Standardsoftware entwickelt. Die Verwaltung wird deutlich einfacher, wenn man das an einen Kunden ausgelieferte Paket als eine Ansammlung lose gekoppelter, für sich eigenständiger Komponenten betrachtet, die flexibel miteinander kombiniert und somit zu einem Produkt zusammengestellt werden können.

Anforderungen an die Architektur

Diese Vorgehensweise stellt spezifische Anforderungen an die Architektur und den Aufbau einer Anwendung. In der Microsoft-Welt bietet sich das MVVM-Pattern [MVVM] in Verbindung mit dem Managed Extensibility Framework [MEF] an. Dies ermöglicht die flexible Zusammenstellung einer Anwendung aus lose gekoppelten Komponenten, wie in Abbildung 2 zu sehen ist.

Hierzu stellen die einzelnen Komponenten (Dienste und Module) in einem Katalog durch Exporte Funktionalitäten bereit und fordern über Importe andere Dienste oder Module an. Das auf dem MEF-basie-

rende Anwendungsframework analysiert die Abhängigkeiten und fügt die Komponenten, wie in Abbildung 3 gezeigt, zu einer Anwendung zusammen.

Zum Beispiel stellt die Klasse *ReportingService*, wie in Listing 1 gezeigt, mithilfe eines *Export-Attributs* einen Dienst vom Typ *IReportingService* bereit.

Listing 1:
Bereitstellung eines MEF-Exports

```
[Export(typeof(IReportingService))]
public class ReportingService :
IReportingService
{
  // IReportingService Members
}
```

Die Anwendung fordert durch Setzen eines Import-Attributs einen Dienst vom Typ IReportingService an (vgl. *Listing 2*).

Listing 2:
Anforderung eines MEF-Imports

```
[Export]
public class Application
{
   [Import (typeof(IReportingService)]
   private IReportingService ReportingSer-
   vice { get; set; }
}
```

Ein Bootstrapper erstellt beim Start einen Katalog. In diesen werden, wie in Listing 3 dargestellt, über einen DirectoyCatalog alle Assemblies, die sich im Ausführungsverzeichnis befinden, aufgenommen.

Listing 3:
Komposition der Anwendung

```
public partial class Bootstrapper
{
  public Bootstrapper()
  {
    var catalog = new
    DirectoryCatalog(Environment.
    CurrentDirectory);
    var container = new
    CompositionContainer(catalog);
    container.ComposeParts(this);
  }
}
```

Über den MEF-*CompositionContainer* werden die sich im Katalog befindenden Exports und Imports zusammengeführt. Eine Anpassung des Katalogs ermöglicht eine andere Zusammenstellung der Anwendung.

Struktur der Quellverwaltung

Die Betrachtung der Dienste und Module als eigenständige Komponenten spiegelt, wie in Abbildung 4 dargestellt, sich in der Quellcodeverwaltung wieder.

Jede Komponente wird als ein separates Produkt behandelt. Die Entwicklung erfolgt auf dem Main- oder den Development-Branches der jeweiligen Komponente. Freigegebene Versionen befinden sich unterhalb des Release-Verzeichnisses. In

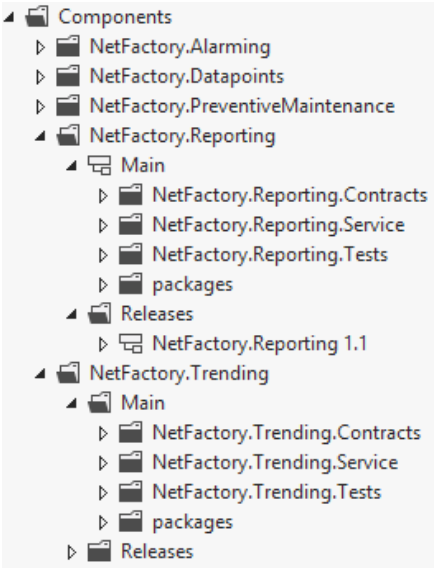

Abb. 4: Behandlung von Komponenten als einzelne Produkte

dieses werden ggf. auch Bugfixes für ältere Versionen eingepflegt. Zwischen den Komponenten bestehen keine direkten Verlinkungen auf Quellcodeebene.

Verwaltung der Komponenten und ihrer Abhängigkeiten

Im Microsoft-Ökosystem hat sich die Open-Source-Paketverwaltung NuGet [NuG] weitgehend etabliert. Auch in die eigenen Entwicklungsprozesse lässt sich NuGet sehr gut einbinden. Die sich im Release-Verzeichnis befindenden Versionen werden über einen Build Server auf einem internen NuGet Package Server [NPS] be-

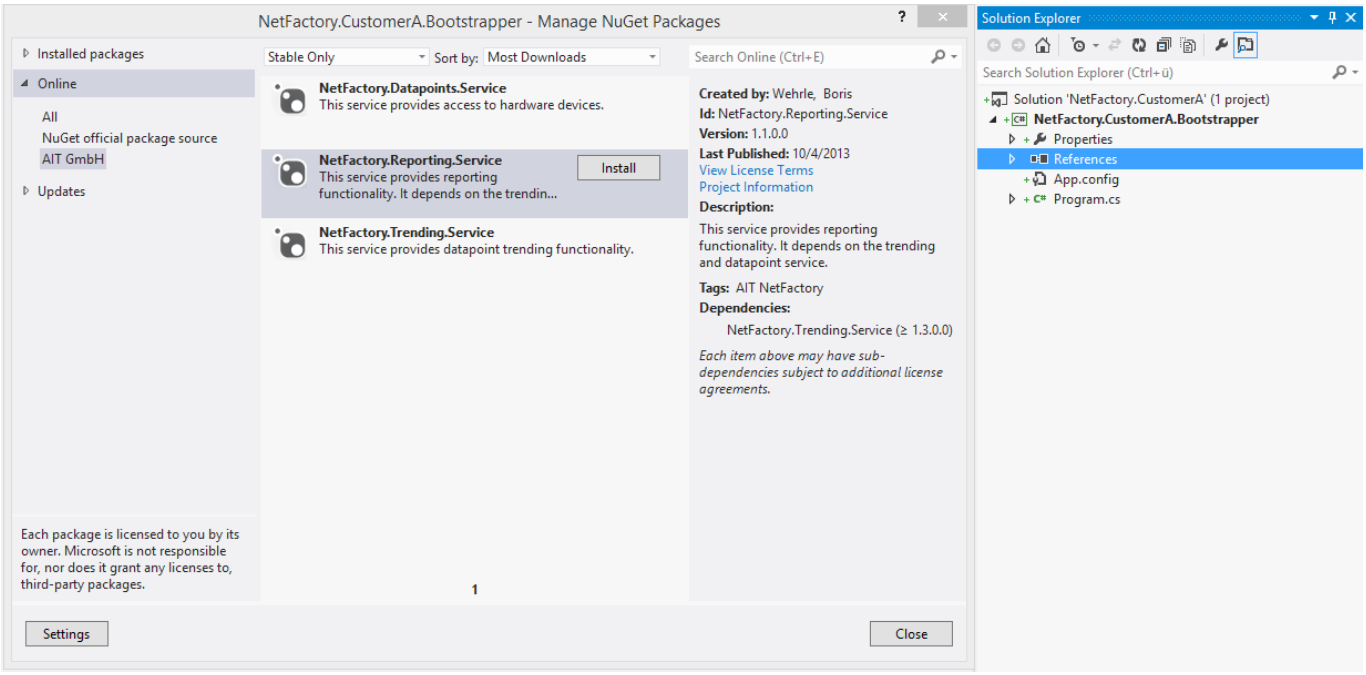

Abb. 5: Bereitgestellte Komponenten auf dem NuGet Package Server

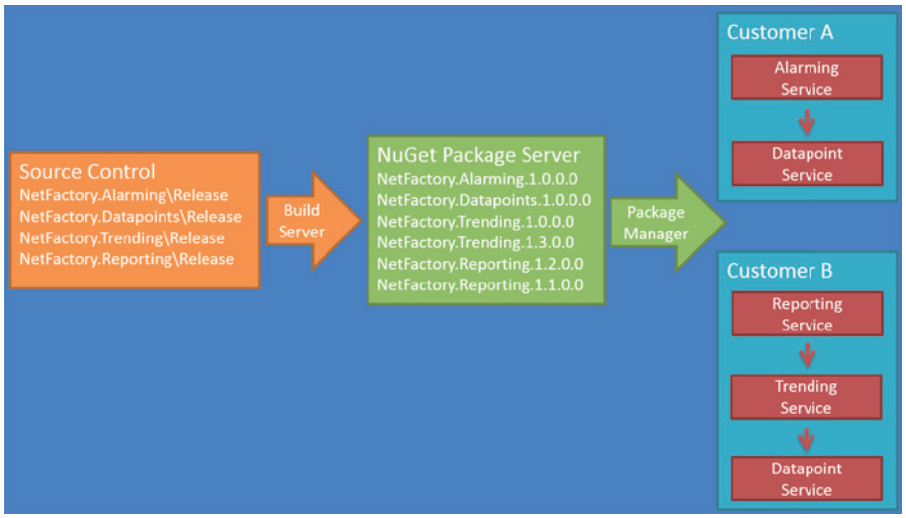

Abb. 6: Zusammenstellung der kundenspezifischen Lösungen

reitgestellt. Auf diesen kann über den in das Visual Studio integrierten NuGet Package Manager, wie in Abbildung 5 zu sehen, zugegriffen werden.

Komponenten, die auf andere Komponenten zurückgreifen, referenzieren diese als NuGet-Pakete. Dies ermöglicht eine automatische Integration von abhängigen Diensten und Modulen.

Zusammenstellung von Komponenten zu Kundenlösungen

Die sich auf dem NuGet Package Server befindenden Komponenten können nun kundenindividuell zusammengestellt werden (siehe Abbildung 6).

Hierzu wird für jeden Kunden eine separate Projektmappe erstellt. Um den Aufwand zu reduzieren, empfiehlt sich die Erstellung einer entsprechenden Projektvorlage. Die Kundenlösung wird als eigenständiges Produkt behandelt. Die Entwicklung bzw. die Zusammenstellung der Komponenten erfolgt auf dem jeweiligen Main Branch, der für den jeweiligen Kunden angelegt wurde und damit unabhängig von den Branches der Komponenten. Der aktuelle Kundenstand wird im Releases Branch bereitgestellt (siehe Abbildung 7).

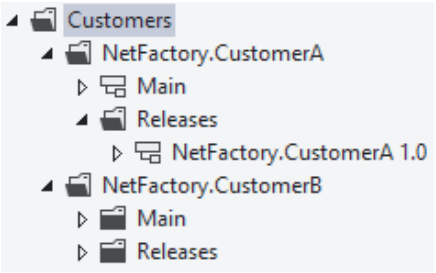

Abb. 7: Abbildung von Kundenlösungen in der Quellcode-Verwaltung

Dieses Vorgehen erlaubt die Zusammenstellung von individuellen Paketen für jeden Kunden. Zusätzlich können in der Struktur ggf. spezielle Dienste und Module für jeden einzelnen Kunden entwickelt werden.

Konfiguration von Komponenten und Erweiterungen

Das Verhalten der zusammengestellten Komponenten kann zusätzlich über eine Konfigurationsdatei angepasst werden. Diese wird im kundenspezifischen Zweig unter Versionsverwaltung gestellt. Zusätzlich können in den Diensten und Modulen Erweiterungspunkte vorgesehen werden. Die Komponente prüft hierbei, ob sich entsprechende Erweiterungen im MEF-Katalog befinden und lädt dies nach. Somit kann z. B. die Reporting-Komponente leicht um kundenspezifische Berechnungsfunktionalitäten erweitert werden ohne die Komponente selbst anpassen zu müssen.

Sicherstellung der Funktionalität

Bei der Weiterentwicklung von Komponenten muss auf die Kompatibilität mit kundenspezifischen Lösungen geachtet werden. Diese vorgenommenen Konfigurationen oder integrierte Erweiterungspunkte könnten von Änderungen betroffen sein. Aus diesem Grund ist es sinnvoll, in die kundenspezifische Projektmappe entsprechende Integrations- bzw. Systemtests zu integrieren, die deren korrektes Verhalten sicherstellen.

Fazit

Kundenindividuelle Lösungen schaffen Wettbewerbsvorteile. Um diese kostengünstig umsetzen zu können, ist eine weitgehende Standardisierung notwendig. Eine Architektur, basierend auf konfigurierbaren und lose gekoppelten Komponenten, schafft die Voraussetzung, um diesen Spagat zu meistern. Das MVVM-Pattern in Verbindung mit MEF bereiten hierfür die Basis. Kombiniert man dies mit einer Verwaltung der Dienste und Module in einem NuGet Package Server erhält man die Flexibilität, die Anforderungen zu meistern ohne dabei im Verwaltungschaos zu versinken. ■

Literatur

[MVVM] Model View ViewModel (MVVM), http://bit.ly/tyrKGt
[MEF] Managed Extensibility Framework (MEF), http://bit.ly/uAaTGY
[NuG] NuGet, http://nuget.org
[NPS] Nuget Package Server, http://bit.ly/17cBzj9