.NET-Komponenten in großen Geschäftsanwendungen

Komponentenbasierte Softwareentwicklung auf der .NET-Plattform (Teil 1)

von Alexander Ramisch und Burkhard Perkens-Golomb

Der erste Teil dieser Artikelserie schildert Erfahrungen bei der Anwendung plattformunabhängiger Entwurfsideen unter .NET. Es werden praxiserprobte Hinweise zur Erstellung eigener Komponenten und Tipps zur Verwendung fremder Komponenten gegeben.

Bevor es ins Detail geht, kurz etwas zur Entstehungsgeschichte dieser Artikelserie. Die *sd&m AG* [1] konzipiert und entwickelt Softwaresysteme für und gemeinsam mit ihren Kunden. Hierbei entstehen in einer Laufzeit von oft mehreren Jahren in Teams von mehreren Entwicklern große Softwaresysteme. Im Gegensatz zu einer Anwendung, die von einer einzelnen Person entwickelt wird, spielen bei großen Systemen Themen wie Modularisierung, Integration oder das Einbinden von Fremdsystemen und -komponenten eine wesentliche Rolle. Ein über Jahre hinweg entstandenes Softwaresystem muss flexibel an die sich im Laufe der Entwicklung ändernden Anforderungen angepasst werden können – es werden also hohe Erwartungen an die Wartbarkeit und Erweiterbarkeit gestellt. Die hier vorgestellten Konzepte und Ideen sind alle in Projekten erprobt.

Komponentenorientierung oder komponentenbasierte Architekturen sind heute gängige Begriffe im Bereich der Softwarearchitektur und -entwicklung. Auch wenn man sich über die Definition des Begriffs *Komponente* nicht immer einig ist, stimmen alle darin überein, dass die Komponentenidee wichtig ist und einen großen Beitrag dazu liefert, die zuvor genannten Anforderungen an Softwaresysteme zu erfüllen. Im Kontext von .NET wird der Komponentenbegriff mehrfach mit unterschiedlicher Bedeutung verwendet, wie folgende Beispiele zeigen:

- Controls auf Dialogen wie z.B. Schaltflächen, Optionsfelder oder Eingabefelder werden als *visuelle Komponenten* bezeichnet.
- *.NET-Assemblies* sind als kleinste Auslieferungseinheit auf der .NET-Plattform die Komponenten jeder .NET-Anwendung – physikalisch liegen sie entweder als EXE- oder DLL-Dateien vor.
- *COM-Komponenten* waren die „Strukturierungseinheit" vor .NET. Auch heute spielen sie noch eine Rolle: So können Teile existierender Anwendungen in neue .NET-Systeme eingebunden

kurz & bündig

Inhalt

Welche Rolle spielen Komponenten in großen Anwendungen und wie werden sie konzipiert und umgesetzt?

Zusammenfassung

Nur weil ein Begriff häufig verwendet wird, bedeutet das nicht, dass er auch klar definiert ist – der erste Teil der vierteiligen Serie holt dies beim Komponentenbegriff nach

Was ist eine Komponente?

In der Literatur findet man zahlreiche Definitionen für den Begriff „Komponente" (siehe z.B. [2][3][4]). Aus Sicht von *sd&m* fehlt bei diesen Definitionen aber noch ein wichtiger Aspekt, der bei den folgenden Komponentenmerkmalen als Eigenschaft Nr. 6 aufgeführt ist:

1. Eine Komponente exportiert eine oder mehrere Schnittstellen. Jede Komponente, die eine Schnittstelle S exportiert, ist eine Implementierung von S.
2. Sie kann andere Schnittstellen importieren. Der Import einer Schnittstelle bedeutet, dass die Komponente die Methoden dieser Schnittstelle benutzt. Sie ist aber erst dann lauffähig, wenn Implementierungen aller importierten Schnittstellen zur Verfügung stehen – das ist Aufgabe der Konfiguration.
3. Sie versteckt die Implementierung und kann daher leicht durch andere Komponenten ersetzt werden, die dieselbe Schnittstelle exportieren.
4. Sie eignet sich als Einheit der Wiederverwendung, denn sie kennt nicht die Umgebung, in der sie läuft, sondern trifft nur minimale Annahmen darüber.
5. Komponenten können andere enthalten oder anders ausgedrückt: Man kann neue Komponenten aus vorhandenen Komponenten zusammensetzen.
6. Nach dem Verständnis der Autoren ist eine Komponente zudem eine softwaretechnische Einheit des Entwurfs, der Implementierung und der Planung [5].

Im Zusammenhang mit modernen Komponentenarchitekturen wird der Komponentenbegriff noch enger gefasst. Eine wesentliche Eigenschaft einer Komponente ist hier die Anpassbarkeit an unterschiedliche Systemumgebungen durch Konfiguration, also ohne die Komponente neu übersetzen zu müssen. Die fachlichen Komponenten in individuell erstellten betrieblichen Informationssystemen erheben selten einen Anspruch auf Wiederverwendbarkeit, da ihr Anteil an individueller Funktionalität sehr hoch ist.

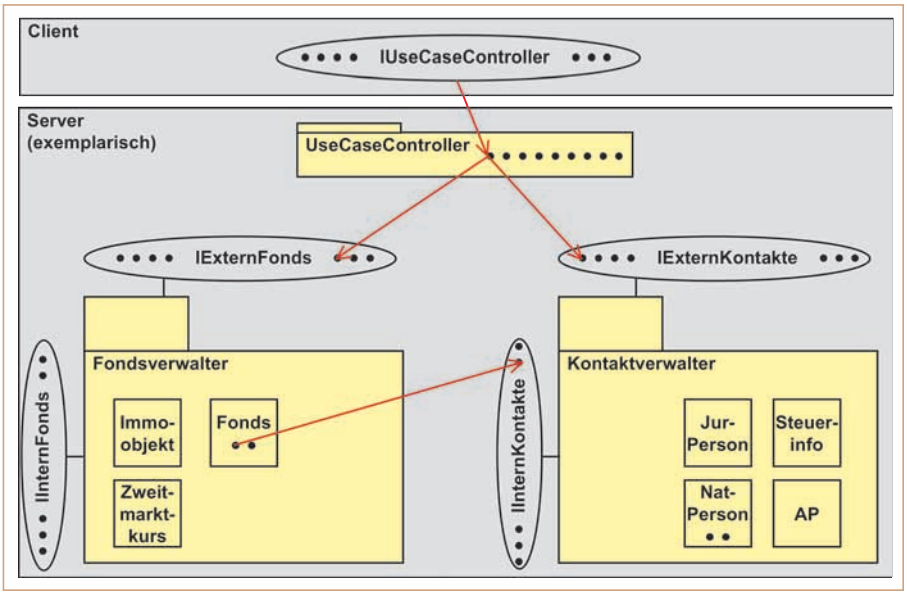

Abb. 1: Entkopplung der Clients vom Server

(mittels COM-Interop) und später ggf. schrittweise gegen neue .NET-Komponenten ausgetauscht werden (z.B. Integration von Microsoft Office).

- In den von Microsoft zur Verfügung gestellten Klassen des .NET Framework findet sich der Namespace *System.ComponentModel* – dieser besteht aus Klassen, die zur Design- oder Laufzeit einer Anwendung komponentenorientierte Funktionalität und Dienste bereitstellen.

.NET-Komponenten und Assemblies

Die aufgeführten Eigenschaften von Komponenten (siehe Kasten „Was ist eine Komponente?“) machen deutlich, dass bei der Entwicklung eines .NET-Systems darauf geachtet werden muss, Schnittstellen und deren Implementierung klar zu trennen. Der Nutzer einer Komponente darf die Interna der genutzten Komponente nicht kennen müssen, nur deren Schnittstelle. Daraus ergeben sich zwei Forderungen:

1) Die Implementierung von einer .NET-Komponente sollte getrennt von ihrer Schnittstelle ausgeliefert werden – Schnittstellen und Implementierungen werden in separaten Assemblies erstellt.
2) Die zur Implementierung des „Komponentengeheimnisses“ verwendeten Klassen dürfen außerhalb der Assembly nicht sichtbar sein – stattdessen sind höchstens deren Schnittstellen bekannt.

Diese Forderungen bedeuten in der Praxis, dass man sich bei einem Client/Server-System sowohl um einzelne Schnittstellen als auch um die transitive Hülle der in den Signaturen der Schnittstellen vorkommenden Typen Gedanken machen muss. Daher müssen alle Abhängigkeiten berücksichtigt werden, die aus verwendeten Interfaces und deren Abhängigkeiten zu weiteren Komponenten resultieren. Folgenden Fragen sind dabei wichtig:

- Was passiert mit Exceptions am Server?
- Sollen diese auch am Client sichtbar sein?
- Wie transportiert man fachliche Daten zwischen Client und Server?

Um Abhängigkeiten vom Client zum Server zu reduzieren, kann man alle Exceptions im Anwendungskern am Server abfangen und diese über geeignete Transportobjekte mit Fehlernummer und einer Meldung an den Client übertragen. Solche Objekte kommen bei der Übertragung fachlicher Objekte zwischen Client und Server oft zum Einsatz – sie definieren als Datencontainer die zu transportierenden Inhalte der Anwendungsobjekte und reduzieren damit die Abhängigkeiten auf wenige simple Containerklassen.

Kopplung zwischen Client und Server über Schnittstellen

Die dargestellten Forderungen sollen nun an einem Beispiel verdeutlicht werden: Es wurde ein Client/Server-System mit Rich-Client-Funktionalität und Fachlichkeit auf einem Anwendungsserver erstellt. Die vom Client aus angestoßenen

Anwendungsfälle werden von Komponenten des Anwendungskerns auf dem Server bearbeitet. Zur Entkopplung der Clients von der Implementierung der Anwendungsfunktionalität am Server kennen sie nur die Schnittstellen, die die Komponenten des Anwendungskerns explizit nach außen hin anbieten. Für die Softwareverteilung bedeutet das, dass die externen Schnittstellen der fachlichen Komponenten am Server in eigenen Assemblies (DLLs) auf die Clients verteilt werden müssen. Die Implementierung der zentralen Komponenten sowie deren Schnittstellen, die sie nur anderen zentralen Komponenten am Server zur Verfügung stellen, werden dagegen nicht an die Clients verteilt [7]. Abbildung 1 macht deutlich, dass fachliche Komponenten des zentralen Anwendungskerns für Clients andere Schnittstellen zur Verfügung stellen als für andere zentrale Komponenten.

Warum Komponenten?

Die Strukturierung eines Systems in Komponenten entspricht der Standardarchitektur von *sd&m*. Diese ist unabhängig von der Technologie, mit der sie implementiert wird, und legt fest, aus welchen Bausteinen oder Komponenten sich das System zusammensetzt [6]. Durch die Verwendung von Komponenten ergeben sich folgende Vorteile:

- Es können Softwareteile von Spezialisten gekauft und eingesetzt werden, z.B. zum Logging, zur Anzeige und Bearbeitung von Grafiken oder für die Darstellung und Bearbeitung komplexer Tabellen.

Was ist eine Schnittstelle?

Wie der Komponentenbegriff besitzt auch der Begriff der Schnittstelle gleich mehrere Bedeutungen. Im Gegensatz zur Benutzerschnittstelle eines Systems geht es hier um die Programmierschnittstelle einer Komponente oder Klasse. Diese Schnittstellen legen fest, welche Funktionalität die Klasse oder Komponente ihren Nutzern bietet. Ein in C# geschriebenes Interface definiert über konkrete Signaturen der angebotenen Methoden die Aufrufssyntax und damit auch die Rückgabewerte, Argumente und *in/out*-Typen. Neben dem Programmcode des Interfaces gehören auch folgende Informationen zur Schnittstelle:

- Die Semantik (was bewirkt die Methode?)
- Das Protokoll (z.B. synchron, asynchron)
- Die nicht funktionalen Eigenschaften wie Performance, Robustheit, Verfügbarkeit etc.

Im Gegensatz zu einem C++-Header, in dem die Schnittstelle und ggf. Implementierungsdetails einer Klasse festgelegt werden, trennen Interfaces explizit die interne Implementierung von der nach außen angebotenen Funktionalität. Ein einfaches Beispiel verdeutlicht dies: Die Schnittstelle *IKonto* definiert die beiden Operationen *Einzahlen* und *Abheben* eines Kontos:

```
interface IKonto
{
 void Einzahlen( float Betrag );
 void Abheben( float Betrag );
}
```

Angenommen, es gäbe zwei unterschiedliche Implementierungen dieser Schnittstelle in Form eines Geschäftskontos und eines Privatkontos. Beide implementieren die Schnittstelle *IKonto* auf ihre eigene Weise und verbergen diese Implementierung:

```
class Geschäftskonto : IKonto
{
 void Einzahlen(float betrag) { ... }
 void Abheben  (float betrag) { ... }
}

class Privatkonto : IKonto
{
 void Einzahlen(float betrag) { ... }
 void Abheben  (float betrag) { ... }
}
```

Bei Verwendung von Komponenten ist es wichtig, gegen Schnittstellen und nicht gegen deren Implementierung zu realisieren – nur so erreicht man eine lose Kopplung und erzielt damit Austauschbarkeit. Als Beispiel soll die Methode *Überweisung* einer Komponente dienen, in der nur die Schnittstelle *IKonto* und nicht die Konkretisierung Geschäfts- oder Privatkonto verwendet werden soll:

```
void Ueberweisung( IKonto a, IKonto b, float betrag )
{
 a.Abheben( betrag );
 b.Einzahlen( betrag );
}
```

Die Stelle, an der für eine Schnittstelle eine konkrete Klasse eingesetzt wird, bezeichnet man als Konfiguration. Hier gibt es verschiedene Möglichkeiten, die Konkretisierung auf bestimmte Implementierungen zu verstecken. Letztlich müssen aber irgendwo im Quellcode konkrete Klassen instanziiert werden, wie

```
IKonto a = new Geschäftskonto();
IKonto b = new Privatkonto();
```

Wer definiert die Schnittstelle?

Das Beispiel zeigte die Verwendung einer vom Exporteur angebotenen Schnittstelle. Es ist auch denkbar, dass der Importeur sich auf die Funktionalität einer anderen Komponente stützt, deren Schnittstelle er selbst festlegt. Die Schwierigkeit besteht darin, einen passenden Exporteur zu finden bzw. diesen genau auf die geforderte Schnittstelle hin zu entwickeln. Sollen Exporteur und Importeur unabhängig voneinander entwickelt werden, nutzen die Autoren das Konstrukt der „Stützschnittstelle“: Der Importeur stützt sich auf einer selbst geschriebenen angeforderten Schnittstelle eines gewünschten Exporteurs ab. Der Exporteur bietet seine Dienste in Form einer angebotenen Schnittstelle an. Bei der Konfiguration werden die Unterschiede der angeforderten und angebotenen Schnittstelle über einen Adapter ausgeglichen.

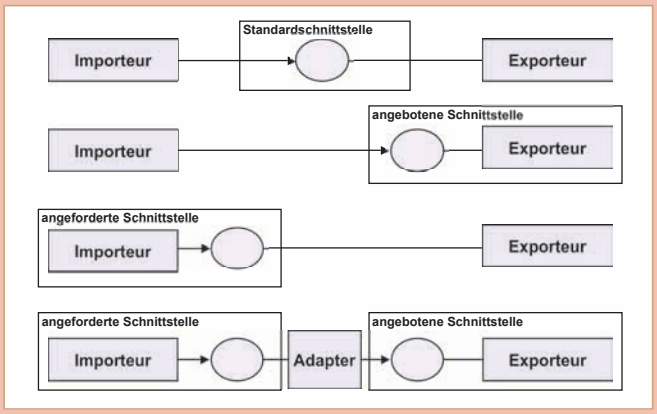

Abb. 2: Die Verantwortung für Schnittstellen kann je nach Typ beim Importeur, Exporteur oder „dazwischen“ liegen

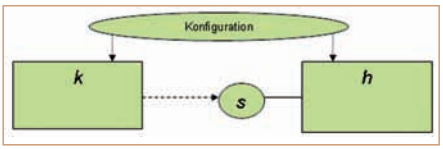

Abb. 3.: Bei der Konfiguration werden zu den genutzten Schnittstellen Implementierungen bereitgestellt

- Komponenten bilden bereits während der Entwicklung voneinander separierte Arbeitspakete, die parallel entwickelt werden – so können Abhängigkeiten minimiert werden.
- Auch nach der Implementierung und Auslieferung bietet ein in Komponenten aufgeteiltes System noch Vorteile: Komponenten erleichtern während einer Wartung die Änderbarkeit der Software. Wenn nur die Schnittstellen einer Komponente bekannt sind, kann die Implementierung der Komponente beliebig geändert werden, solange die vereinbarte Schnittstelle gleich bleibt. Somit kann ein Softwaresystem an einer Stelle korrigiert und erweitert werden, ohne die Gefahr, dass es sich danach an anderer Stelle nicht mehr korrekt verhält.

Die Komponenten in betrieblichen Informationssystemen verstecken fachliche oder technische Klassen, die die Aufgabe der Komponente erfüllen. Diese Klassen sind außerhalb der Komponente nur über Schnittstellen bekannt und werden darum in C# mit der Sichtbarkeit *internal* realisiert. Damit sind sie auch technisch nur innerhalb der Assembly der Komponente selbst verwendbar.

Listing 1

Die Komponente *Zentrale Dienste*

```
public class ZentraleDienste
{
  // Der Zusicherungsdienst
  private IAssert _assertPlugin;

  // Registrieren eines Zusicherungsdienstes
  public static void RegisterAssertPlugin
                                    ( IAssert assertPlugin )
  {
    _instance._assertPlugin = assertPlugin;
  }

  // Zugriff auf das Zusicherungs-Plugin
  public static IAssert Assert
  {
    get{ return _instance._assertPlugin; }
  }
  ...
}
```

Wie finde ich die Komponenten meines Systems?

Es gibt keinen Algorithmus, der ein System in Komponenten aufteilt. In der Literatur wird eine Vielzahl von Anhaltspunkten diskutiert, wie man bei einem frühen Systementwurf zu geeigneten Komponenten kommt. Die Erfahrung der Autoren zeigt, dass es sich hierbei um einen iterativen Prozess handelt. Ein zunächst grober Schnitt wird immer weiter verfeinert. Dabei betrachtet man gefundene Komponenten immer wieder genauer, um sie ggf. in weitere Komponenten zu zerlegen oder bestehende zusammenzufassen. Komponenten mit vielen Querbeziehungen und enger Kopplung fasst man besser zusammen, so erhält man eine größere Komponente mit einem starken inneren Zusammenhalt im Gegensatz zu zwei Komponenten, die sehr eng miteinander in Beziehung stehen (somit gibt es weniger Abstimmungsaufwand zwischen den beiden Komponentenverantwortlichen, weniger Projekte, …). Ein guter Komponentenschnitt ist schließlich an einer kurzen einprägsamen Beschreibung der gewählten Aufteilung erkennbar.

Konfiguration

Konfiguration kann zweierlei bedeuten: Im .NET Framework enthält der Namespace *System.Configuration* Klassen, die es einer Anwendung erlauben, Konstanten aus einer XML-Datei zu lesen. Die Auslagerung von Konstanten in eine externe XML-Datei, anstatt sie in den Code der Anwendung zu integrieren, erlaubt es, die Anwendung später leicht umzukonfigurieren. Im Folgenden wird die Konfiguration eines komponentenbasierten Systems betrachtet, die dafür verantwortlich ist, die bestehenden Komponenten zu einem lauffähigen System zusammenzusetzen. Dabei werden konkrete Implementierungen zu den im System verwendeten Schnittstellen festgelegt. Es handelt

Listing 2

```
// In diesem Beispiel wird aus der Komponente Fonds
// heraus ein Kontakt gelesen
// Verwalter der Komponente Kontakt erfragen
IKontaktverwalter KontaktVerwalter =
      Komponentenverwalter.Instanz.IKontaktverwalter;

// gewünschten Kontakt lesen
IPerson = KontaktVerwalter.Lesen( "Mustermann,Michael" );
```

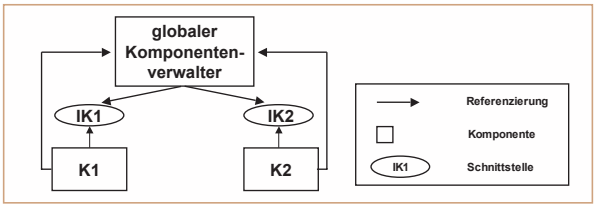

Abb 4: Zugriff auf Komponenten über Verwalter

sich also um die Zuordnung von importierter Schnittstelle zu implementierender Komponente – der Zeitpunkt der Konfiguration kann dabei variieren. Denkbar sind folgende Varianten:

- bei der Übersetzung des Programms (*early binding* zur Compile-Zeit)
- beim Programmstart oder Aufruf einer Komponente (*late binding* zur Laufzeit)

Wichtig ist, dass die Komponenten selbst durch die Konfiguration nicht verändert werden. Abbildung 3 zeigt die Komponentennutzung über eine Schnittstelle, die durch die Konfiguration konkretisiert wird. Diese verbindet dabei die beiden Komponenten *k* und *h* über die Schnittstelle *s*.

Die Konfiguration ermöglicht es, Teile eines bestehenden Systems auszutauschen oder ein System aus bestehenden Bausteinen zusammenzustecken – sie ist also ein Mittel zur Wartbarkeit und Erweiterbarkeit. Darüber hinaus ermöglicht die Technik der Konfiguration Tests in schlanker Umgebung, weil eine zu testende Komponente in das Testsystem integriert werden kann, ohne dass die Komponente davon erfährt. Nach erfolgreichen Tests kann die Komponente dann ohne Änderung des Sourcecodes in das Produktivsystem eingeklinkt werden. Für weitere Informationen zu unterschiedlichen Varianten der privaten und öffentlichen Konfiguration wird auf die Ausführungen in [5] verwiesen.

Komponentenverwaltung

Eine Komponente besteht meist aus mehreren Klassen, die für den Komponentennutzer nicht oder höchstens über Interfaces bekannt sind. Nun stellt sich die Frage, wie auf eine Komponente zugegriffen werden kann. Es muss eine Klasse geben, die den Zugriff auf die Komponente bietet – diese bezeichnen die Autoren auch als Komponentenverwalter, der die zur Erbringung des Komponentendienstes notwendigen Klassen erzeugen, löschen und finden kann. Für den Zugriff auf Dienste einer Komponente sind verschiedene Lösungen denkbar. Die beiden nun folgenden Varianten *Plug-in-Architektur* und *Komponentenverwalter als zentrale Registry* kamen bereits in verschiedenen Systemen zum Einsatz.

Plug-in-Architektur

Bei dieser Architektur wird eine zentrale Klasse festgelegt, die Komponentenaufrufe an die registrierten *Plug-ins* (andere Komponenten) weiterleitet. Diese Methode kann angewendet werden, wenn eine Komponente beispielsweise für querschnittliche Dienste benötigt wird, die in verschiedenen Projekten zum Einsatz kommen und der Zugriff auf diese einheitlich stattfinden soll. Man kann sich vorstellen, dass in unterschiedlichen Projekten jeweils verschiedene Implementierungen mit derselben Schnittstelle für Zusicherungen eingesetzt werden. Der in den Projekten entstehende Anwendungscode greift stets in gleicher Weise auf die Komponente für zentrale Dienste zu. Welche konkrete Implementierung beispielsweise für Zusicherungen hinter den Kulissen den Dienst erbringt, kann durch Konfiguration an einer zentralen Stelle über das Plug-in festgelegt werden. So können die Projekte unterschiedliche Komponenten für denselben Zweck nutzen ohne die Nutzung selbst variabel implementieren zu müssen. Die Schnittstelle einer Zusicherungskomponente könnte z.B. so aussehen:

```
public interface IAssert
{
 void Assert( bool condition, string message );
 void AssertNotNull( object anObject, string message);
 ...
}
```

Beim Betrachten einer mit Plug-in-Architektur entworfenen Komponente *ZentraleDienste* (Listing 1) für querschnittliche Dienste wie Tracing, Logging oder Assert finden sich folgende Bestandteile:

- Ein Member auf der Schnittstelle *IAssert* einer Assert-Implementierung
- Ein statisches Property zum Zugriff auf die hinterlegte Implementierung
- Eine statische Methode zum Registrieren einer konkreten Implementierung für den Vorgang des Einklinkens (Plug-in)

Im Anwendungscode wird auf die registrierten querschnittlichen Dienste wie Zusicherungen ausschließlich über die Komponente *ZentraleDienste* zugegriffen. Die Konfiguration, also das mittels Plug-in durchzuführende Registrieren der konkreten Implementierungen, erfolgt an zentraler Stelle beispielsweise beim Start der Anwendung. Hier kann die zu verwendende Implementierung entweder auskodiert sein oder aus einer Config-Datei gelesen werden. Zusicherungen werden im Anwendungscode wie folgt verwendet:

```
ZentraleDienste.Assert.AssertParamNotNull
                    ( bValue, "Beschreibung zum Fehler");
```

Komponentenverwalter als zentrale Registry

Eine andere Variante wäre ein zentraler Komponentenverwalter, der wiederum

Listing 3

Ausschnitt der Komponente Fonds

```
namespace leonardo.server.awk.fonds
{
 public class Fondsverwalter : IKAwkFondsExtern,
                               IKAwkFondsIntern
 {
  private Fondsverwalter(){}

  // RegistrierenInstanz ist eine spezielle
  // Implementierung des Singleton-Patterns
  // (getInstance), bei dem die statische Methode
  // in diesem Fall nicht das Singleton zurückliefert,
  // sondern es im  Fondsverwalter verfügbar macht
  public static void RegistrierenInstanz()
  {
   if( Fondsverwalter.Instanz == null )
    Fondsverwalter.Instanz = new Fondsverwalter();

   Komponentenverwalter.Instanz.IKAwkFondsExtern =
    Fondsverwalter.Instanz;

   Komponentenverwalter.Instanz.IKAwkFondsIntern =
    Fondsverwalter.Instanz;
  }
 }

 internal class Fonds : IFonds
 {
  //...
 }
}
```

die Verwalter der im System genutzten Komponenten kennt. Ein Verwalter ist dabei eine spezielle Klasse pro Komponente, die den Zugriff auf die Komponente ermöglicht. Verwalter bieten typischerweise Methoden zur Erzeugung, Änderung und Löschung sowie zur Suche von Entitäten. Dieser Verwalter registriert sich selbst bei einer global bekannten und als Singleton realisierten Verwalterklasse (in Abbildung 4 der globale Komponentenverwalter), die die Schnittstellen der einzelnen Verwalter jeder Komponente kennt.

Als Beispiel soll die Komponente *Kontakt* aus einem Client/Server-System dienen. Sie enthält die Klasse *Kontaktverwalter*, die die Schnittstellen *IInternKontakte* und *IExternKontakte* implementiert. Hier sind Methoden enthalten, mit denen Kontakte angelegt, gesucht, gelöscht etc. werden können. Wenn aus einer beliebigen Komponente, wie z.B. *Fonds*, auf einen Kontakt zugegriffen werden muss, wird im Quellcode zunächst die Schnittstelle des Kontaktverwalters vom globalen Komponentenverwalter erfragt, um anschließend an diesem den Zugriff auf die Kontakte durchzuführen (Listing 2).

Zugriff auf Komponenten

Im Initialisierungsprozess wird die Anwendung konfiguriert, indem die einzelnen Komponentenverwalter erzeugt werden und sich mit ihren Schnittstellen am globalen Komponentenverwalter registrieren. Die Komponenten dieses Client/Server-Systems verfügen meist über zwei verschiedene Schnittstellen: Eine interne mit Diensten, die am Server verfügbar sind, und eine externe mit Diensten, die auch am Client bekannt sind. Das Codebeispiel in Listing 3 zeigt einen Ausschnitt der Komponente *Fonds*. Hier sieht man den Verwalter der Komponente, der die beiden Schnittstellen *IAWKFondsExtern* und *IAWKFondsIntern* implementiert. Am Fondsverwalter dient die statische Methode *RegistrierenInstanz()* dazu, die öffentlichen Schnittstellen am globalen Komponentenverwalter einzutragen – sie werden dazu als Property gesetzt.

Inversion of Control mittels Dependency Injection

Während der Komponentenverwalter als zentrale Registry zur Compile-Zeit oder beim Programmstart einmal initialisiert wird und danach eine statische Komponentenverwaltung darstellt, können bei der Plug-in-Variante Implementierungen zur Laufzeit ausgetauscht werden. Beide vorgestellten Ansätze erfordern, dass alle verwalteten Komponenten die Registry kennen. Somit besteht von allen verwalteten Komponenten aus eine Abhängigkeit zum Komponentenverwalter, der wiederum alle Komponenten kennt. *Martin Fowler* beschreibt, wie man mittels *Dependency Injection* die Abhängigkeit von Komponenten zu geeigneten Implementierungen ohne eine zentrale Komponentenverwaltung realisiert [7]. Eine Komponente erfragt bei diesem Ansatz nicht die benötigte Implementierung einer anderen importierten Komponente von einem Verwalter, sondern erhält diese über einen Parameter im Konstruktor oder über den Aufruf eines Setters injiziert – somit wird der Kontrollfluss invertiert (*Inversion of Control*, kurz *IOC*). Nach dem Dependency-Injection-Muster gibt es eine separate Komponente, die für die Konfiguration sorgt und die benötigten Implementierungen der Komponenten zur Verfügung stellt. Weitere Details zu Dependency Injection und den Varianten der Realisierung unter zu Hilfenahme von existierenden Frameworks wie *Spring, Pico Container, HiveMind* und *XWork* sind unter [8] zu finden.

Alexander Ramisch und Burkhard Perkens-Golomb arbeiten bei sd&m als Software-Entwickler.

Links & Literatur

[1] www.sdm.de

[2] D.F. D'Souza, A.C. Wills: Objects, Components and Frameworks with UML: The Catalysis Approach, Addison Wesley 1999

[3] C. Szyperski: Component Software, Addison Wesley 2002

[4] OMG: Unified Modelling Language Specification 2.0.2004: www.uml.org

[5] Johannes Siedersleben: Moderne Software Architektur, dpunkt.verlag 2004

[6] Alexander Ramisch: Standardarchitektur für .NET Systeme: www.sdm.de/de/it-wissen/themen/dotnet/

[7] Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern: www.martinfowler.com/articles/injection.html

[8] Nene, Dehananjay: A beginners guide to dependency injection, Juli 2005: www.theserverside.com/articles/article.tss?l=IOCBeginners