# .NET im Einsatz

## Komponentenbasierte Software-Entwicklung mit .NET in großen Geschäftsanwendungen, Teil 2

von Alexander Ramisch und Burkhard Perkens-Golomb

Dieser Artikel schildert Erfahrungen bei der Anwendung plattformunabhängiger Entwurfsideen unter .NET – es werden praxiserprobte Hinweise zur Erstellung eigener Komponenten und Tipps zur Verwendung fremder Komponenten gegeben.

Obwohl Komponentenorientierung und komponentenbasierte Architekturen heute gängige Begriffe sind, gibt es doch immer wieder unterschiedliche Vorstellungen bezüglich ihrer Bedeutung. Aus diesem Grund haben die Autoren die bei der *sd&m AG* [1] verwendete Definition für die Begriffe Komponente und Schnittstelle im ersten Teil dieser Artikelserie [2] definiert und anhand von Beispielen verdeutlicht. Nachdem der Theorieteil absolviert ist, geht es in dieser Folge um ein praktisches Beispiel aus dem Projektgeschäft von sd&m.

### Komponentenschnitt anhand eines praktischen Beispiels

Als Beispiel aus der Praxis soll das inzwischen bei der Münchener Rück produktiv eingesetzte Anwendungssystem mit dem Namen *PRICE* dienen (der Entwicklungsaufwand liegt inzwischen bei über 25 Bearbeiterjahren), das auf Basis von Eingabedaten umfangreiche statistische Analysen und auch Prognoserechnungen durchführt. Technisch gesehen handelt es sich um ein auf .NET basierendes Rich-Client-System: Ein zentraler Server mit Web-Service-Schnittstelle bietet Dienste zur Datenhaltung. Weltweit verteilte, mit Windows-Forms realisierte Clients erbringen sowohl die Anwendungs- als auch die Präsentationslogik. Sie fordern die fachlichen Daten vom zentralen Server an und sorgen für deren Verarbeitung und Darstellung. Die mathematische Funktionalität der Anwendung ist von einem anderen Entwicklerteam in *Unmanaged* C++-Objekten implementiert, die durch *Managed* C++-Wrapper eingebunden werden. Die Clients bedienen sich zum Importieren und Exportieren von Daten u.a. Microsoft-Excel via COM-Interop und integrieren Komponenten anderer Hersteller.

Verschiedene Aspekte der Anwendung von Komponententechnik werden an dieser Anwendung sichtbar: Die Aufteilung der Anwendungsfunktionalität in Einzelkomponenten, Integration von COM-Komponenten (Excel) und anderer Produkte sowie Einbindung einer „Legacy"-Komponente implementiert in unmanaged C++. Im Folgenden werden die Wahl des Komponentenschnitts und die Integration der Legacy-Komponente näher betrachtet

### Aufteilung der Anwendung in logische Komponenten

Die Anwendung besitzt eine typische 3-Schicht-Architektur (neben in allen Schichten genutzter Querschnittsfunktionalität). Die Funktionalität jeder einzelnen Schicht und die Querschnittsfunktionalität wurden nach fachlichen Gesichtspunkten und technischen Aufgabenstellungen in Komponenten aufgeteilt (Komponenten für die Excel-Schnittstelle, für den Zugriff auf ein zentrales Autorisierungssystem, für die statistischen Berechnungen usw.). Diese Komponenten bieten nach außen nur Schnittstellen (Interfaces) für Objekte und den Komponentenverwalter an. Jeder Komponentennutzer, der konkrete Implementierungen einer Schnittstelle verwenden möchte, muss sich an den Verwalter der Komponente wenden und erhält von diesem Objekte mit der gewünschten Schnittstelle. Von diesen Objekten ist dem Nutzer somit nur die Schnittstelle, aber nicht die konkrete Implementierung bekannt (wie es auch im ersten Teil dieser Artikelserie [2] im Absatz „Komponentenverwaltung" beschrieben wurde). Pro Schicht und für die gesamte Querschnittsfunktionalität gibt es jeweils einen zentralen Verwalter, der die Verwalter der einzelnen Komponenten kennt. Ein solcher „Oberverwalter" ist als Singleton realisiert und bietet Zugang zu den Einzelkomponenten (vgl. Absatz „Komponentenverwalter als zentrale Registry" im ersten Teil [2]).

**kurz & bündig**

**Inhalt**

Welche Rolle spielen Komponenten in großen Anwendungen und wie werden sie konzipiert und umgesetzt? Diese mehrteilige Serie gibt Empfehlungen aus Praxisprojekten, an deren Umsetzung die Autoren beteiligt waren

**Zusammenfassung**

Die reine Lehre, nach der jede Klasse und jede Schnittstelle in getrennten Assemblies untergebracht werden, führt zu einem deutlich aufwändigeren Build-Prozess, sodass es ähnlich wie bei der Normalisierung von Tabellen auf den den goldenen Mittelweg ankommt

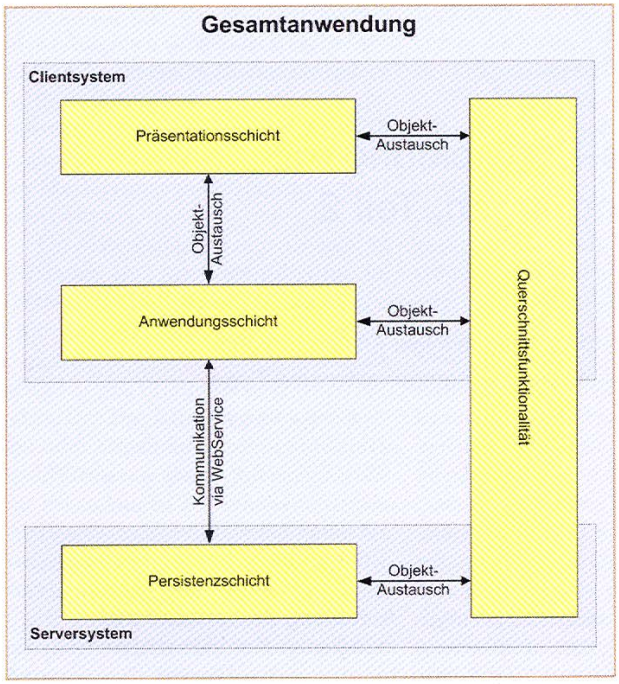


Abb. 1.:Schematische Darstellung der 3-Schicht-Architektur des Beispielsystems

Ein Beispiel für eine Komponente der Anwendung ist die versicherungsmathematische Statistikfunktionalität bei PRICE. Die Schnittstelle eines Objekts dieser Komponente enthält Methoden zur Durchführung von Berechnungen (Listing 1). Die Schnittstelle des Verwalters (Listing 2) bietet Zugang zu den Implementierungen der Schnittstellen. Die Schnittstellen *IStatisticalFunctions* und *IMathematicalFunctionsManager* haben entsprechende Implementierungen (dabei ist sowohl die Klasse *StatisticalFunctions* als auch ihr Konstruktor als *internal* deklariert und somit nicht außerhalb der Komponente sichtbar – siehe Listing 3). Beim Start der Anwendung wird bei einer zentralen Verwalterklasse *GeneralManager* (Listing 4) der Komponentenverwalter der mathematischen Funktionalität während der Konfigurationsphase (z.B. zu Beginn der *Main*-Methode) registriert. Ein Nutzer der Komponente besorgt sich vom zentralen Verwalter eine Referenz auf den Komponentenverwalter und ruft darüber die Dienste der Komponente auf (Listing 5).

Die Vorteile dieses Vorgehens: Die Komponenten eines Systems kennen sich nur über Schnittstellen, Interna sind gegenseitig unbekannt, sodass die einzelnen Komponenten losgelöst voneinander entwickelt und getestet werden können. Da mit dem Verwalter einer Komponente ein singulärer Zugang zu einer Komponente existiert, können Komponenten leicht ausgetauscht werden. Existiert z.B. eine Alternativimplementierung der Statistikfunktionalität (mit alternativen Algorithmen, Leerimplementierungen für Testzwecke usw.), zu sehen in Listing 6, muss zu deren Benutzung nur die Registrierung des Komponentenverwalters geändert werden, die in der Konfigurationsphase der Anwendung stattfindet:

```
MathematicalFunctionsManager2.RegisterInstance();
```

Auf diese Weise können mit minimalem Aufwand ganze Komponentenimplementierungen ausgetauscht werden.

## Logische Komponenten vs. physische Komponenten

In einer ersten Version von PRICE wurden die physischen Komponenten (.NET

**Listing 1**

```
namespace MathematicalFunctions
{
 // Schnittstelle für statistische
 // Berechnungsfunktionen.
 public interface IStatisticalFunctions
 {
  object CalculateStatisticalFunctionABC(...);
  object CalculateStatisticalFunctionXYZ(...);
  ...
 }
}
```

**Listing 2**

```
namespace MathematicalFunctions
{
 // Schnittstelle des Verwalters der Komponente
 // mathematischer Funktionen.
 public interface IMathematicalFunctionsManager
 {
  IStatisticalFunctions CreateStatisticalFunctions();
  ...
 }
}
```

**Listing 3**

```
namespace MathematicalFunctions
{
 // Implementierung der Schnittstelle
 // IStatisticalFunctions.
 internal class StatisticalFunctions : IStatisticalFunctions
 {
  internal StatisticalFunctions() {}
  public object CalculateStatisticalFunctionABC(...){}
  public object CalculateStatisticalFunctionXYZ(...){}
  ...
 }

 // Implementierung der Schnittstelle
 // IMathematicalFunctionsManager.
 public class MathematicalFunctionsManager :
                              IMathematicalFunctionsManager
 {
  private MathematicalFunctionsManager() {}

  // Registriert diesen Komponentenverwalter beim
  // "Überverwalter". Wird in der Konfigurationsphase
  // der Anwendung gerufen.
  public static void RegisterInstance()
  {
   GeneralManager.Instance.
                        MathematicalFunctionsManager =
    new MathematicalFunctionsManager();
  }

  ...
  public IStatisticalFunctions CreateStatisticalFunctions()
  {
   return new StatisticalFunctions();
  }
  ...
 }
}
```

Assemblies) entsprechend den logischen Komponenten entworfen: Die Schnittstellen einer Komponente wurden einem Visual-Studio-Projekt und die diese Schnittstellen implementierenden Klassen einem weiteren Projekt zugeordnet. Damit werden von Visual Studio .NET pro Komponente, bestehend aus Schnittstellen und Implementierung, je zwei Assemblies erzeugt. Diese Vorgehensweise bedeutete maximale Entkopplung aller Schnittstellen und Implementierungen und damit auch die Möglichkeit maximaler Parallelisierung der Entwicklung der einzelnen Komponenten – aber leider auch größtmöglicher Verwaltungsaufwand für die Entwicklungsumgebung.

Die Verwaltung von Solutions mit einer großen Zahl von Projekten (50 und mehr) erwies sich in Visual Studio .NET als recht problematisch. Allein das Öffnen einer solchen Solution nimmt auch bei guter Rechnerausstattung mehrere Minuten in Anspruch und ein Rebuild aller Projekte wird zum Geduldsspiel. Abhilfe schafft die Reduktion der Projektanzahl einer Solution durch den Umstieg von Projekt- auf File-Referenzen oder die Aufgabe des Prinzips „Komponente = Schnittstellen-Assembly + Implementierungs-Assembly". Um den deutlich komplexeren Build-Prozess und den zusätzlichen Aufwand für die Versionsführung bei Verwendung von File-Referenzen zu vermeiden, wurde der zweite Weg gewählt. Eine Assembly der Anwendung enthält demnach typischerweise viele Komponenten einer Schicht des Systems. Konkret wurden bei PRICE jeweils alle Schnittstellen aller Komponenten einer Schicht zu einer Assembly und alle Implementierungsklassen aller Komponenten einer Schicht zu einer weiteren Assembly zusammengefasst.

**Listing 4**

Die Verwalterklasse GeneralManager

```
MathematicalFunctionsManager.RegisterInstance();


public class GeneralManager
{
 // Singleton-Instanz.
 public static readonly GeneralManager Instance =
                                  new GeneralManager();

 // Privater Konstruktor
 private GeneralManager() {}

 private IMathematicalFunctionsManager _
                                  mathematicalFunctionsManager;

 // Zugang zum Komponentenverwalter der Komponente
 //der mathematischen Berechnungen.
 public IMathematicalFunctionsManager
                                  MathematicalFunctionsManager
 {
  get { return _mathematicalFunctionsManager; }
  set { _mathematicalFunctionsManager = value; }
 }
 ...
}
```

**Listing 5**

```
IMathematicalFunctionsManager mathematical
        FunctionsManager = GeneralManager.Instance.
                        MathematicalFunctionsManager;

IStatisticalFunctions statisticalFunctions =
 mathematicalFunctionsManager.
                        CreateStatisticalFunctions();

statisticalFunctions.CalculateStatisticalFunctionABC(...);
statisticalFunctions.CalculateStatisticalFunctionXYZ(...);
++
```

Die Reduktion der Anzahl von Assemblies vermindert auch den Verwaltungsaufwand der CLR zur Laufzeit und erlaubt dem JIT-Compiler weitergehende Optimierungen, denn dieser führt Performanceverbesserungen wie Inlining nicht über Assembly-Grenzen hinweg durch. Je kleiner die Anzahl und somit größer der Umfang der Assemblies, umso größer die Gewinne durch Optimierungen des JIT-Compilers. Dieses Beispiel zeigt, dass es bei der Realisierung von logischen Konzepten zu Abweichungen von der „reinen Lehre" kommen kann. Ähnlich wie bei der Denormalisierung von relationalen Datenmodellen wird vom Architekten die notwendige Balance von gutem Design und Praxistauglichkeit gefordert. Eines darf man dabei aber nicht vergessen: Durch die Zusammenfassung der Komponenten einer Schicht zu einer Assembly geht die Semantik der Zugriffsbeschränkung mittels *internal* verloren. Der Entwickler einer Komponente *A* in der Schicht *X* kann also auf beliebige Implementierungen anderer Komponenten der Schicht *X* zugreifen. Somit hat man keine Überprüfung auf Ebene der Entwicklungsumgebung, sondern muss entsprechende Regeln in den Programmierrichtlinien im Team vereinbaren und in Reviews überprüfen.

**Listing 6**

**Eine alternative Verwalterklasse**

```
namespace MathematicalFunctions
{
 // Alternative Implementierung der Schnittstelle
IStatisticalFunctions.
  //internal class StatisticalFunctions2 :
  //IStatisticalFunctions
 {
  internal StatisticalFunctions2() {}
  public object CalculateStatisticalFunctionABC(...){}
  public object CalculateStatisticalFunctionXYZ(...) {}
 }

 // Alternative Implementierung der Schnittstelle
 //IMathematicalFunctionsManager
 public class MathematicalFunctionsManager2 :
                        IMathematicalFunctionsManager
 {
  private MathematicalFunctionsManager2() {}

  // Registriert diesen Komponentenverwalter beim
  // "Überverwalter". Wird in der Konfigurationsphase
  // der Anwendung gerufen.
  public static void RegisterInstance()
  {
   GeneralManager.Instance.
                        MathematicalFunctionsManager=
    new MathematicalFunctionsManager2();
  }

  public IStatisticalFunctions CreateStatisticalFunctions()
  {
   return new StatisticalFunctions2();
  }
 }
}
```

## Integration einer Legacy-Komponente

Die zuvor als Beispiel angeführte Komponente für die Nutzung der Statistikfunktionalität ist auch aus anderen Gründen interessant: Sie führt die mathematischen Berechnungen nicht selbst durch, sondern stützt sich auf eine weitere Komponente, die von einem separaten Teilteam in *Unmanaged* C++ entwickelt wird. Der in *Unmanaged* C++ programmierte Teil der Anwendung ist ebenfalls nach den Grundsätzen des Komponentenaufbaus gestaltet: Von allen Objekten ist nach außen hin nur eine Schnittstelle (C++-Klassen mit ausschließlich abstrakten Methoden) bekannt, konkrete Implementierungen bleiben „Geheimnis" der Komponente. Objekte, die die Schnittstellen implementieren, werden ausschließlich von einem Verwalter der Komponente erzeugt. Die Integration dieser besonderen, weil getrennt entwickelten und technisch anders gearteten Komponente erfolgte wie bei einer typischen Legacy-Komponente: Es wird eine zusätzliche neue Komponente mit eigenen Schnittstellen und eigenem Verwalter eingeführt, deren einzige Aufgabe die Kapselung des Legacy-Teils ist.

Die neue Komponente übernimmt vor allem die Zustandsverwaltung der C++-Objekte: Die Berechnungsfunktionen arbeiten nämlich nicht wie eine simple Funktionsbibliothek, bei der alle Methoden zu jedem beliebigen Zeitpunkt aufgerufen werden können. Vielmehr bauen die teilweise sehr umfangreichen Berechnungsschritte aufeinander auf und müssen in einer festgelegten Reihenfolge gerufen werden. Oder anders ausgedrückt: Die C++-Objekte halten einen inneren Zustand, der den Aufruf bestimmter Methoden abhängig von diesem inneren Zustand nur zu gewissen Zeitpunkten erlaubt. Die Kapselkomponente verbirgt das Wissen um Zustände und Aufrufreihenfolge – an ihr kann bei Bereitstellung entsprechender Eingabedaten das Ergebnis eines jeden Rechenschrittes zu jedem beliebigen Zeitpunkt angefordert werden. Intern führt die Kapselkomponente Buch, welche Daten an die C++-Objekte übergeben und welche Rechenschritte durchgeführt werden. Wird eine Rechenmethode an der Kapselkomponente aufgerufen, vergleicht sie die neuen Eingabedaten mit den bereits an die C++-Objekte übertragenen Daten. Daraus wird abgeleitet, welche Daten noch an die C++-Objekte übertragen und welche Rechenschritte von der nicht verwalteten Komponente gegebenenfalls wiederholt bzw. neu ausgeführt werden müssen, um zum gewünschten Ergebnis zu gelangen.

Die Schnittstelle der Kapselkomponente verbirgt vor dem Aufrufer auch, wo die *Unmanaged* C++-Objekte leben: Zum einen können diese *Unmanaged* Objekte im selben Prozess wie der gesamte Client existieren. Zu Diagnosezwecken oder um die GUI von möglichen Abstürzen des nativen C++-Teils abzuschirmen, kann für die *Unmanaged* Objekte und ihre *Managed* Wrapper ein separater Prozess gestartet werden, der mit dem Client per Remoting kommuniziert. Der Implementierungsaufwand für die Kapselkomponente ist hoch, wird aber durch den erzielten Gewinn gerechtfertigt: Die Legacy-Komponente ist vom Rest des Anwendungssystems soweit wie möglich entkoppelt, sämtliche Interna über die Verwaltung der *Unmanaged* C++-Komponente sind für den Aufrufer der Kapsel-Komponente verborgen. Ebenso wie dadurch im Anwendungssystem Belange getrennt sind, kann im Entwicklerteam das Wissen um die Implementierungsdetails der Kapsel auf einen kleinen Kreis konzentriert sein. Die Entkopplung der Komponenten ermöglicht die Parallelisierung ihrer Entwicklung und führt zu besserer Wartbarkeit.

**Alexander Ramisch** arbeitet bei sd&m als Seniorberater und **Burkhard Perkens-Golomb** bei der Münchener Rück als IT-Architekt.

### Links & Literatur

[1] www.sdm.de

[2] Alexander Ramisch und Burkhard Perkens-Golomb: .NET-Komponenten in großen Geschäftsanwenungen Teil 1, in: *dot.net magazin* 3.2006