.NET im Einsatz

Komponentenbasierte Software-Entwicklung in großen Geschäftsanwendungen, Teil 3

von Burkhard Perkens-Golomb und Alexander Ramisch

Der dritte Teil dieser Artikelserie gibt Hinweise zur Implementierung komponentenorientierter Konzepte unter .NET – mithilfe der wenig bekannten Klassen aus dem Namespace System.ComponentModel.

Komponenten müssen geschnitten, ihre Außensicht muss definiert werden – diese Problemstellungen waren Gegenstand der ersten zwei Teile dieser Artikelserie ([1] + [2]), die komponentenorientierte Konzepte von einer eher plattformunabhängigen Warte betrachtet haben. Komponenten wollen aber letztlich auch implementiert sein. Wer dies unter .NET tut, muss nicht vollkommen bei „Null" beginnen: Microsoft bietet mit den Klassen des Namespace *System.ComponentModel* viel Unterstützung für typische Aufgaben der komponentenorientierten Programmierung. Die Funktionalität dieser Klassen wird im Designer von Visual Studio oder häufig zur Laufzeit einer Anwendung für Lizensierungsaufgaben genutzt. Selten werden diese Klassen dagegen zur Laufzeit einer Anwendung für komponentenorientierte Funktionalität eingesetzt. Es sollen nun einige interessante Aspekte der möglichen Nutzung herausgegriffen sowie Hinweise und Beispiele zu deren Einsatz gegeben werden. Die Erkundung dieses Namespaces ist eine manchmal doch beschwerliche Reise – aber es lohnt sich, denn es sind dort noch viele Schätze zu entdecken.

kurz & bündig

Inhalt

Im dritten Teil dieser Artikelserie geht es um Typenkonvertierer, Deskriptoren und die Möglichkeit, dass Komponenten Änderungen ihres Zustands über Events anderen Komponenten mitteilen

Zusammenfassung

Insbesondere die Deskriptoren stellen eine wenig bekannte Möglichkeit dar, zur Laufzeit z.B. Property-Informationen zu erweitern und bieten gegenüber Reflection eine Reihe von Vorteilen

Konvertierung von Datentypen mithilfe der TypeConverter

Eine häufig wiederkehrende Aufgabe in Informationssystemen ist die Konvertierung von Datenwerten in unterschiedliche Darstellungen, z.B. für die Benutzeroberfläche (GUI), die Datenbank, eine Druckaufbereitung etc. Werden eigene Datentypen verwendet, wird die notwendige Funktionalität häufig in Methoden der Datentypklassen implementiert (Listing 1). Dieser Ansatz hat gleiche mehrere Nachteile:

- Datentypen sollen nur einfache Container für durch sie repräsentierte Werte sein. Sie werden durch Konvertierungsmethoden mit überflüssiger Funktionalität überfrachtet, ihre Schnittstelle wird aufgebläht. Im Sinne einer guten Komponentenorientierung sollten Implementierungen für Datenhaltung und Wertumwandlung getrennt sein (Trennung der Zuständigkeiten).
- Jeder Aufrufer der Konvertierungsmethoden benötigt eine Instanz der Datentypklasse. Was aber, wenn zur Laufzeit eine *null*-Referenz vorgefunden wird? An allen Aufrufstellen der Umwandlungsmethoden müsste explizit auf eine *null*-Referenz geprüft und dieser Fall gesondert behandelt werden.
- Bei Enum-Klassen versagt der Ansatz, dass Wertobjekte ihre Umwandlung selbst besorgen: Aufzählungen sind in .NET Value-Typen und diese können daher nicht um eigene Methoden erweitert werden.

Ein alternativer Ansatz, der die obigen Nachteile vermeidet, schlummert verborgen im Namespace *System.ComponentModel* in Gestalt der sog. *TypeConverter*. Diese Basisklasse für Repräsentationsumwandlungen definiert eine Schnittstelle, bietet grundlegende Funktionalität und legt ein kleines Programmiermodell für Umwandlungen fest. Soll diese Basisklasse für einen eigenen Datentyp genutzt werden,

Listing 1

Eine Klasse mit Konvertierungsfunktionen

```
public class MyDatatypeWithConvertMethods
{
  /// Standardmethode
  public override string ToString() {...}

  /// Umwandlung für GUI
  public string ToRepresentationForGui() {...}

  /// Umwandlung für Logging
  public string ToRepresentationForLogging() {...}

  /// Umwandlung für Datenbank
  public object ToRepresentationForDB() {...}

  /// Umwandlung für Druckaufbereitung
  public object ToRepresentationForPrinting() {...}

  ...
}
```

geht man wie folgt vor. Für einen eigenen Datentyp *MyDatatype* wird eine Konvertierungsklasse erstellt, die sich von der Basisklasse *TypeConverter* ableitet. Diese Konvertierungsklasse wird spezifisch für diesen Datentyp implementiert und überschreibt die vier Methoden *CanConvertFrom*, *ConvertFrom*, *CanConvertTo* und *ConvertTo* der Basisklasse (Listing 2).

Der Methodenparameter *ITypeDescriptorContext* kann bei Bedarf genutzt werden, um nähere Umstände der Konvertierung zu beschreiben (z.B. Umwandlung wird gerade durchgeführt von oder für GUI, Datenbank oder Druck usw.). Über das Metaattribut *TypeConverterAttribute* an der Klassendefinition des eigenen Datentypen wird die Konverterklasse dem Datentypen zugeordnet:

```
[TypeConverter(typeof(MyDatatypeConverter))]
public class MyDatatype {...}
```

Das Metaattribut an der Datentypklasse macht diese Zuordnung des Konverters einem zentralen Verzeichnisdienst des Namespace *System.ComponentModel* bekannt. Zur Laufzeit bietet die Klasse *System.ComponentModel.TypeDescriptor* den Zugang zu diesem Verzeichnisdienst, dort wird bei Bedarf eine Instanz des Konverters angefragt

```
TypeConverter converterForMyDatatype =
        TypeDescriptor.GetConverter(typeof(MyDatatype));
```

und zur Konvertierung genutzt (Listing 3). Die zuvor genannten Nachteile von Konvertierungsmethoden in Datentypklassen entfallen bei diesem Vorgehen. Die Vorteile gehen aber mit einem etwas erhöhten Schreibaufwand bei der Benutzung eines *TypeConverter* einher, was durch Hilfsmethoden oder -Properties etwas abgemildert werden kann (z.B. ein statisches Property an jeder eigenen Datentypklasse, das den zugehörigen *TypeConverter* liefert). Ähneln sich verschiedene Datentypen, kann auch eine *TypeConverter*-Klasse die Umwandlung für mehrere Datentypklassen übernehmen.

Eine Alternative zu Reflection

Das Konzept der sog. *Deskriptoren* wird anhand des Beispiels eines Grid-Control beschrieben: Ein Grid-Control stellt in der Regel die Zeilen einer Tabelle dar, im einfachsten Fall entspricht eine Grid-Spalte einer Tabellenspalte. Ein Grid kann aber auch eine Liste beliebiger (gleichartiger) Objekte visualisieren, dann werden als Spalten die öffentlichen Properties der Objekte gezeigt. In beiden Fällen ist das Grid hochdynamisch – es stellt Objekte dar, deren Struktur es erst zur Laufzeit erfragen kann. Dementsprechend kann auf Werte der darzustellenden Objekte kein statischer Zugriff erfolgen, bei dem aufzurufende Member bereits zur Compilezeit bekannt sind, sondern der Zugriff erfolgt mithilfe des sog. *late binding* – erst zur Laufzeit wird festgelegt, welches Member eines Objekts aufgerufen wird.

Soll zur Laufzeit einer Anwendung die Gestalt einer Klasse (Methoden, Properties usw.) oder der Zustand eines Objekts (Werte der Properties, der Felder usw.) ermittelt werden, ist Reflection das meist verwendete Mittel. Für Properties und Events gibt es allerdings mit den *Deskriptoren* eine Alternative, die wesentlich mehr Flexibilität bietet: Deskriptoren vermögen fast alles, was entsprechende Reflection-Objekte können, beherrschen auch den *late-binding*-Zugriff und bieten noch einiges mehr. So können unter anderem Aufrufern öffentliche Properties „vorgegaukelt" und existierende Properties „versteckt" werden.

Wie nutzen Grids diese Möglichkeiten? Obwohl Grids in der Regel Tabellenzeilen (z.B. Instanzen von *System.Data.DataRowView*) darstellen, ist man beim Blick in die Implementierung überrascht: Es gibt im Programmcode eines Grids keine eigene Funktionalität zur Darstellung von Tabellen und deren Zeilen, programmiert ist nur die Darstellung von Objekten und deren öffentlicher Properties. Eine Tabellenzeile hat aber keine öffentlichen Properties für die dargestellten Spalten, sondern nur ein einziges indiziertes Property, dem man beim Aufruf den Namen oder Index der gewünschten Spalte mitteilen muss. Die Nutzung eines indizierten Property ist im Grid aber nicht programmiert. Die Lösung: Eine Tabellenzeile simuliert mithilfe von Deskriptoren für jede Spalte ein öffentliches Property – wie das im Detail funktioniert, wird im Folgenden gezeigt.

Property- und Event-Deskriptoren

Wenn Properties, auf die zugegriffen werden soll, erst zur Laufzeit bekannt sind, wird in der Regel die Klasse *PropertyInfo* aus dem Namespace *System.Reflection* bemüht. Hat man z.B. eine Klasse *ClassWithProperty*

```
public class ClassWithProperty
{
 public object MyProperty1
 {
  get {...}
  set {...}
 }
 ...
}
```

Listing 2

Die Klasse leitet sich von der TypeConverter-Klasse ab

```
public class MyDatatypeConverter : TypeConverter
{
 // Rückgabewert zeigt an, ob ein Wert der angegebenen Klasse sourceType in
 // eine Instanz von MyDatatype umgewandelt werden kann.
 public override bool CanConvertFrom( ITypeDescriptorContext context,
                Type sourceType ) {...}

 // Konvertiert übergebenen Wert in eine Instanz von MyDatatype.
 public override object ConvertFrom( ITypeDescriptorContext context,
                CultureInfo culture, object value ) {...}

 // Rückgabewert gibt an, ob eine Instanz von MyDatatype in
 // eine Instanz der angegebenen Klasse destinationType umgewandelt werden kann.
  public override bool CanConvertTo( ITypeDescriptorContext context,
                Type destinationType ) {...}

 // Wandelt eine Instanz von MyDatatype in eine Instanz
 // der angegebenen Klasse destinationType um.
 public override object ConvertTo( ITypeDescriptorContext context,
                CultureInfo culture, object value,
                Type destinationType ) {...}
}
```

kann von deren *Type*-Objekt eine *PropertyInfo*-Instanz für *MyProperty1* angefordert werden (Listing 4). Mithilfe einer solchen *PropertyInfo*-Instanz kann man dynamisch „Getter" oder „Setter" des Property aufrufen:

```
// PropertyInfo für den lesenden Zugriff auf das
// Property nutzen.
object propertyValue = propertyInfo.GetValue
                                (objectWithProperty, null);
```

Es geht aber auch anders: Mit einer Instanz der Klasse *PropertyDescriptor* aus dem Namespace *System.ComponentModel*. Diese erhält man über den zuvor bereits erwähnten Verzeichnisdienst von *System.ComponentModel*, den man mit der Klasse *TypeDescriptor* abfragen kann (Listing 5). Der Zugriff mithilfe einer *PropertyDescriptor*-Instanz ist dem mittels eines *PropertyInfo* Objekts sehr ähnlich:

```
// PropertyDescriptor für den lesenden Zugriff auf das
// Property nutzen.
object propertyValue = propertyDescriptor.GetValue
                                (objectWithProperty);
```

Property-Änderungen lösen Events aus

PropertyInfo und *PropertyDescriptor* repräsentieren beide ein Property und bieten beide einen lesenden und schreibenden Zugriff auf dieses Property. *PropertyDescriptor*-Instanzen haben aber einen bedeutenden Vorteil gegenüber *PropertyInfo*-Objekten: Über die Methode *PropertyDescriptor.AddValueChanged(...)* kann ein *EventHandler*-Delegate angemeldet werden. Wird ein Property über einen Deskriptor modifiziert, werden anschließend die beim Deskriptor angemeldeten Delegates aufgerufen. So können alle interessierten Objekte über Änderungen des Property informiert werden, sofern sie von einem zugehörigen *PropertyDescriptor* durchgeführt werden. Auf diese Weise werden die Änderungen an Objekten beobachtbar, was z.B. Grundvoraussetzung für die Anwendung des bekannten Model-View-Controller-Pattern ist.

Deskriptoren haben gegenüber ihren „Konkurrenten" aus dem Reflection-Lager auch einen Nachteil: Der Aufruf von *TypeDescriptor.GetProperties(...)* liefert einen Container, der im Standardfall nur Repräsentanten für öffentlich sichtbare Properties enthält. *Type.GetProperty(...)* dagegen liefert bei geeigneter Parametrisierung und entsprechenden Rechten der Anwendung auch nicht-öffentliche Properties (dies auch unter Verletzung der kodierten Sichtbarkeiten wie *private* oder *protected*). Beim Vergleich *PropertyInfo* vs. *PropertyDescriptor* stellt sich schnell die Frage, welchen Gewinn als den Benachrichtigungsmechanismus der Einsatz von *PropertyDescriptor*-Objekten bringt? Überraschende Antwort: Durch Deskriptoren alleine erzielt man gar keinen weiteren Gewinn gegenüber dem Einsatz von Reflection. Im Gegenteil – intern bedienen sich die Standard-Deskriptorobjekte (dies ist die interne Klasse *ReflectPropertyDescriptor*) sogar einer *PropertyInfo*-Instanz für den Property-Zugriff, sodass der Deskriptor auf den ersten Blick nur eine relativ funktionsarme Hülle für den Zugriff via Reflection zu sein scheint. Kombiniert man aber Deskriptoren mit Objekten, die das Interface *ICustomTypeDescriptor* implementieren, können öffentliche Properties „simuliert" und im Programmtext kodierte öffentliche Properties „versteckt" werden.

Das Interface ICustomTypeDescriptor

Der Reflection-Aufruf *Type.GetProperties(...)* liefert je nach Parametrisierung alle oder eine Auswahl aller physisch vorhandenen Properties eines Zielobjekts. Der Aufruf von *TypeDescriptor.GetProperties(...)* liefert im Standardfall einen Container mit Deskriptoren, die die öffentlichen Properties des Zielobjekts repräsentieren. Dies kann vom Zielobjekt allerdings geändert werden, indem es das Interface *ICustomTypeDescriptor* implementiert:

```
namespace System.ComponentModel
{
 interface IcustomTypeDescriptor
 {
  ...
  // Liefert die PropertyDescriptor-Instanzen
  // dieses Objekts.
  PropertyDescriptorCollection GetProperties();
  ...
 }
}
```

Besitzt das Zielobjekt die Schnittstelle *ICustomTypeDescriptor*, wird der Aufruf von *TypeDescriptor.GetProperties(...)* an *ICustomTypeDescriptor.GetProperties(...)* delegiert. Die Deskriptoren in der Ergebnismenge, die der letztgenannte Aufruf liefert, sind an keinerlei Regeln gebunden (außer, dass sie von der Basisklasse *PropertyDescriptor* abgeleitet sein müssen):

- Ein solcher Deskriptor kann ein öffentliches Property repräsentieren
- oder aber auch ein nicht-öffentliches Property.
- Repräsentanten für physisch vorhandene (d.h. kodierte) öffentliche Properties können in der Ergebnismenge ausgelassen werden
- und es können *PropertyDescriptor*-Instanzen erscheinen, die gar keine physische Entsprechung haben (sozusagen „virtuelle" Properties).
- Die Deskriptoren können über besondere Eigenschaften verfügen: So können z.B. eigene *PropertyDescriptor*-Implementierungen für den Property-Zugriff

Listing 3

```
MyDatatype myValue = new MyDatatype(...);

// Wandle myValue in String-Form um
String s = converterForMyDatatype.ConvertToString
                                (myValue);

// Wandle myValue in eine Instanz eines anderen
// Datentypen MyOtherDatatype um
MyOtherDatatype fooValue = (MyOtherDatatype)
    converterForMyDatatype.ConvertTo(myValue, typeof
                                (MyOtherDatatype));
```

Listing 4

```
ClassWithProperty objectWithProperty =
                                new ClassWithProperty(...);

// PropertyInfo-Instanz für das Property "MyProperty1"
// besorgen
PropertyInfo propertyInfo =
 objectWithProperty.GetType().GetProperty
                                ("MyProperty1");
```

Listing 5

```
ClassWithProperty objectWithProperty =
                                new ClassWithProperty(...);

// PropertyDescriptor-Instanz für das Property
//"MyProperty1" besorgen ...
PropertyDescriptor propertyDescriptor =
 TypeDescriptor.GetProperties(objectWithProperty)
                                ["MyProperty1"];
```

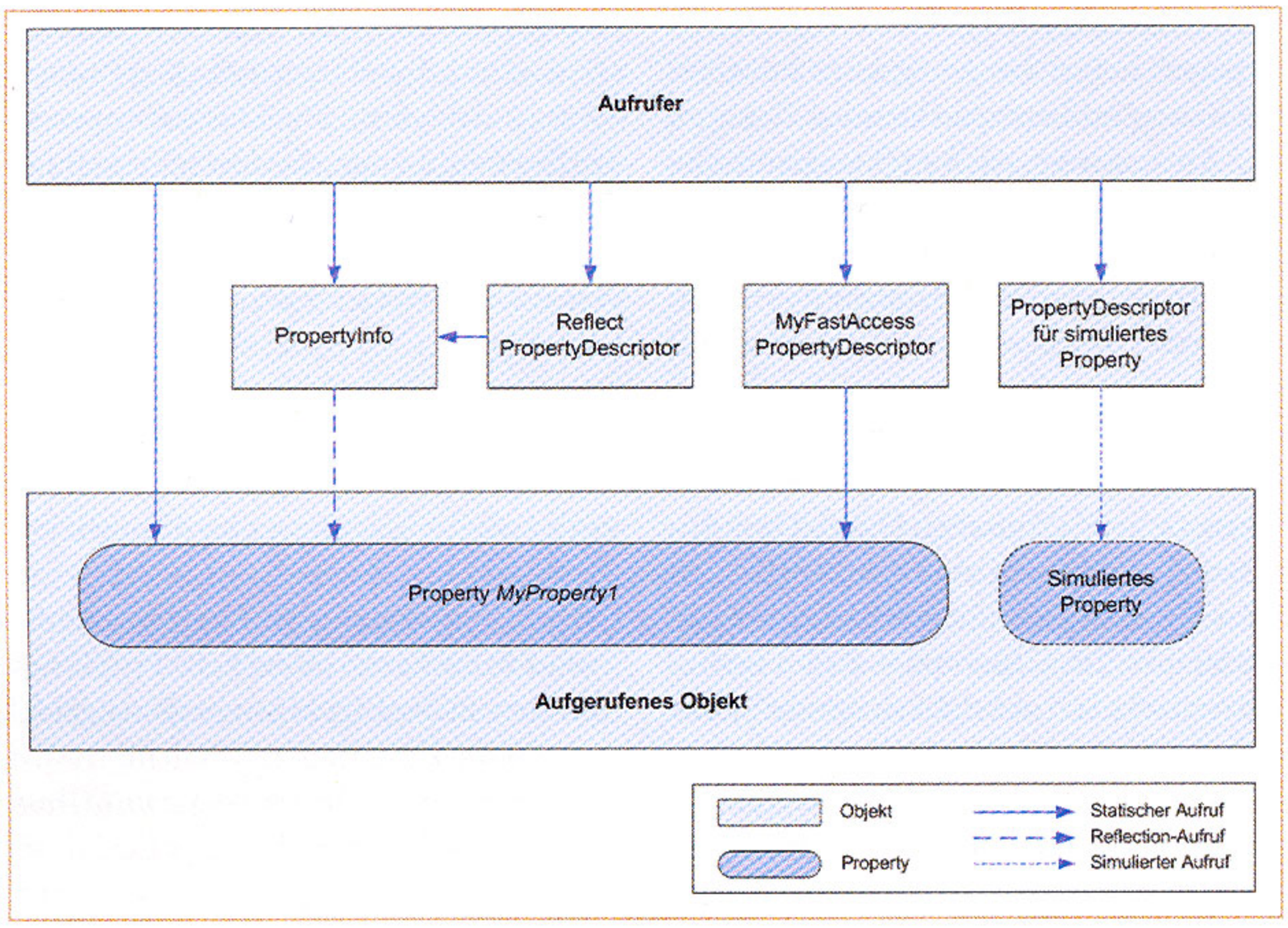

Abb. 1. Schematische Darstellung der verschiedenen Arten eines Property-Zugriffs

auf den generellen Reflection-Ansatz verzichten und stattdessen einen viel schnelleren direkten Zugriff auf das Zielobjekt durchführen.

Das Interface *ICustomTypeDescriptor* bietet somit die Möglichkeit, die Deskriptor-Objekte einer Klasse unabhängig von tatsächlichen Properties nach eigenem Gutdünken zu gestalten. Ein Beispiel: Der auf Reflection basierende Standarddeskriptor für das Property *ClassWithProperty.MyProperty1* soll durch eine schnellere Implementierung ersetzt werden, die das Property *MyProperty1* direkt nutzt. Dazu ist ein eigener Deskriptor zu erstellen, der sich von der Basisklasse *PropertyDescriptor* ableitet und die Methoden *GetValue* und *SetValue* implementiert (Listing 6).

Der neue Deskriptor muss den Standarddeskriptor ersetzen. Hierzu ist die Klasse *ClassWithProperty* so zu modifizieren, dass sie das Interface *ICustomTypeDescriptor* implementiert (Listing 7). Wird jetzt *TypeDescriptor.GetProperties(...)* mit einer Instanz von *ClassWithProperty* als Parameter aufgerufen, so wird dieser Aufruf an die Methode *ClassWithProperty.GetProperties()* weitergeleitet und der Aufrufer erhält als Ergebnis eine Collection mit dem Deskriptor *FastAccessMyProperty1PropertyDescriptor*. Dessen Zugriff auf das Property *MyProperty1* arbeitet deutlich schneller als der auf Reflection basierende Zugriff des Standarddeskriptors.

Deskriptorobjekte

Welchen Vorteil bieten die Deskriptoren eigentlich für die komponentenbasierte Anwendungsentwicklung? Oft werden in einer Anwendung Fremdkomponenten genutzt. Bedient sich diese Fremdkomponente des Reflection-Mechanismus, um auf Objekte außerhalb von ihr zuzugreifen, gibt es kaum Möglichkeiten, den Zugriff nach eigenen Wünschen zu beeinflussen. Benutzt die Fremdkomponente aber Deskriptor-Instanzen, so können die Zielobjekte des Zugriffs ggf. das Interface *ICustomTypeDescriptor* implementieren und damit die für die Fremdkomponente sichtbaren Deskriptoren (und damit quasi auch die für die Fremdkomponente sichtbaren Properties) frei definieren.

Zurück zum Beispiel des Grid-Controls: Programmiert ist im Grid die Anzeige von Objekten und deren öffentlichen Properties. Dennoch können Spalten von Tabellenzeilen dargestellt werden, obwohl es für die Spalten an den Tabellenzeilenobjekten (z.B. Instanzen von *System.Data.DataRowView*) keine öffentlichen Properties gibt. Wie also findet das Grid seine Spalten und Zellwerte? Eine Tabellenzeile gaukelt dem Grid für jede Spalte ein öffentliches Property vor – mithilfe des Interfaces *ICustomTypeDescriptor*. Das Grid fordert nämlich als Repräsentanten für die öffentlichen Properties Deskriptoren mit

Listing 6

Der Standarddeskriptor wird durch einen schnelleren PropertyDescriptor ersetzt

```
public class FastAccessMyProperty1PropertyDescriptor : PropertyDescriptor
{
  // Konstruktor. Ruft den Konstruktor der Basisklasse.
  public FastAccessMyProperty1PropertyDescriptor()
   : base("MyProperty1", new Attribute[] {}) {}

  // Implementiert abstrakte Methode der Basisklasse.
  public override object GetValue(object component)
  {
    // Dieser PropertyDescriptor ist ausschließlich für das Property MyProperty1
    // der Klasse ClassWithProperty zuständig. Daher muss der Parameter
    // component eine Instanz der Klasse ClassWithProperty sein.
    ClassWithProperty objectWithProperty = (ClassWithProperty) component;

    return objectWithProperty.MyProperty1;
  }

  // Implementiert abstrakte Methode der Basisklasse.
  public override void SetValue(object component, object value)
  {
    // Dieser PropertyDescriptor ist ausschließlich für das Property MyProperty1
    // der Klasse ClassWithProperty zuständig. Daher muss der Parameter
    // component eine Instanz der Klasse ClassWithProperty sein.
    ClassWithProperty objectWithProperty = (ClassWithProperty) component;

    objectWithProperty.MyProperty1 = value;

    OnValueChanged(component, EventArgs.Empty);
  }
  ...
}
```

dem Aufruf *TypeDescriptor.GetProperties(...)* an. Stellt das Grid öffentliche Properties eines beliebigen Objekts dar, liefert *TypeDescriptor.GetProperties(...)* genau das gewünschte Resultat: Es werden Deskriptoren für die öffentlichen Properties zurückgegeben. Im Falle einer Tabellenzeile werden die Deskriptoren für das Grid vom Tabellenzeilenobjekt selbst erzeugt, da es das Interface *ICustomTypeDescriptor* implementiert. Diese speziellen Deskriptoren sehen aus wie solche von öffentlichen Properties, in Wirklichkeit aber reräsentieren sie die einzelnen Spalten der Tabellenzeile (der Name dieser „virtuellen" Properties ist dabei gleich dem Spaltennamen). Dem Grid ist das eine wie andere recht: Es benutzt einfach die Deskriptoren, um an die darzustellenden Werte zu gelangen, ob die Deskriptoren nun eine physische Entsprechung im dargestellten Objekt haben oder nicht.

Ein Beispiel aus der Praxis

Die erste und bereits produktive Version des im zweiten Teil dieser Artikelserie [2] erwähnten betrieblichen Informationssystems *PRICE* arbeitet mit einer dateibasierten Speicherung der Anwendungsdaten. Die Daten werden dabei mit einem *BinaryFormatter* in einen *FileStream* geschrieben. Die Folgeversion der Anwendung sollte zusätzlich mit einer datenbankgestützten Datenhaltung versehen werden, wobei eine vom Kunden entwickelte Persistenzkomponente eingesetzt werden musste. Da die erste Version von *PRICE* ohne Kenntnis der Persistenzkomponente entworfen wurde, gab es Inkompatibilitäten zwischen den Entitätsobjekten und den Erwartungen der Persistenzkomponente an zu speichernde Objekte:

- Die Persistenzkomponente geht davon aus, dass **alle** öffentlichen Properties eines Entitätsobjekts persistiert werden. Auf die Properties wird von der Persistenzkomponente zur Laufzeit mittels Reflection zugegriffen. Die Speicherung aller öffentlichen Properties ist bei den vorhandenen Entitätsobjekten von *PRICE* allerdings nicht gewünscht, da es sich bei einigen z.B. um Ergebnisse von Rechnungen oder Verdichtungen handelt, die zur Laufzeit der Anwendung immer aktuell berechnet werden müssen.
- Die Persistenzkomponente erwartet einige nicht fachlich, aber technisch notwendige Properties an den Entitätsobjekten (Schlüsselattribute, Attribute zur Versionsführung etc.). Diese technischen Properties müssten in allen bestehenden Entitätsobjekten nachprogrammiert werden. Aber weder ist der zusätzliche Aufwand wünschenswert, noch sind die technischen Properties für alle anderen Systemteile außer der Persistenzkomponente von Nutzen. Die Schnittstelle der Entitätsobjekte würde also unnötig vergrößert. Funktionalität für die Verwaltung von Aggregationen und Assoziationen ist in den Entitätsobjekten zwar bereits vorhanden, aber in der vorhandenen Form für die Persistenzkomponente nicht nutzbar, sodass diese Funktionalität gedoppelt werden müsste.

Listing 7

```
public class ClassWithProperty : ICustomTypeDescriptor
{
  ...
  public object MyProperty1
  {
    get {...}
    set {...}
  }
  ...
  public PropertyDescriptorCollection GetProperties()
  {
    PropertyDescriptor descriptor =
        new FastAccessMyProperty1PropertyDescriptor();

    PropertyDescriptor[] allDescriptors =
                    new PropertyDescriptor[] {descriptor};

    return new PropertyDescriptorCollection(allDescriptors);
  }
  ...
}
```

Die Lösung liegt in der Verwendung von *PropertyDescriptor*-Objekten durch die Persistenzkomponente und der Implementierung des Interfaces *ICustomTypeDescriptor* durch die Entitätsobjekte:

- Die Persistenzkomponente wurde so modifiziert, dass sie statt Reflection jetzt Deskriptoren für den Zugriff auf Entitätsobjekte verwendet. Der Änderungsaufwand war gering, da sich Aufrufsyntax von *PropertyInfo* und *Pro-*

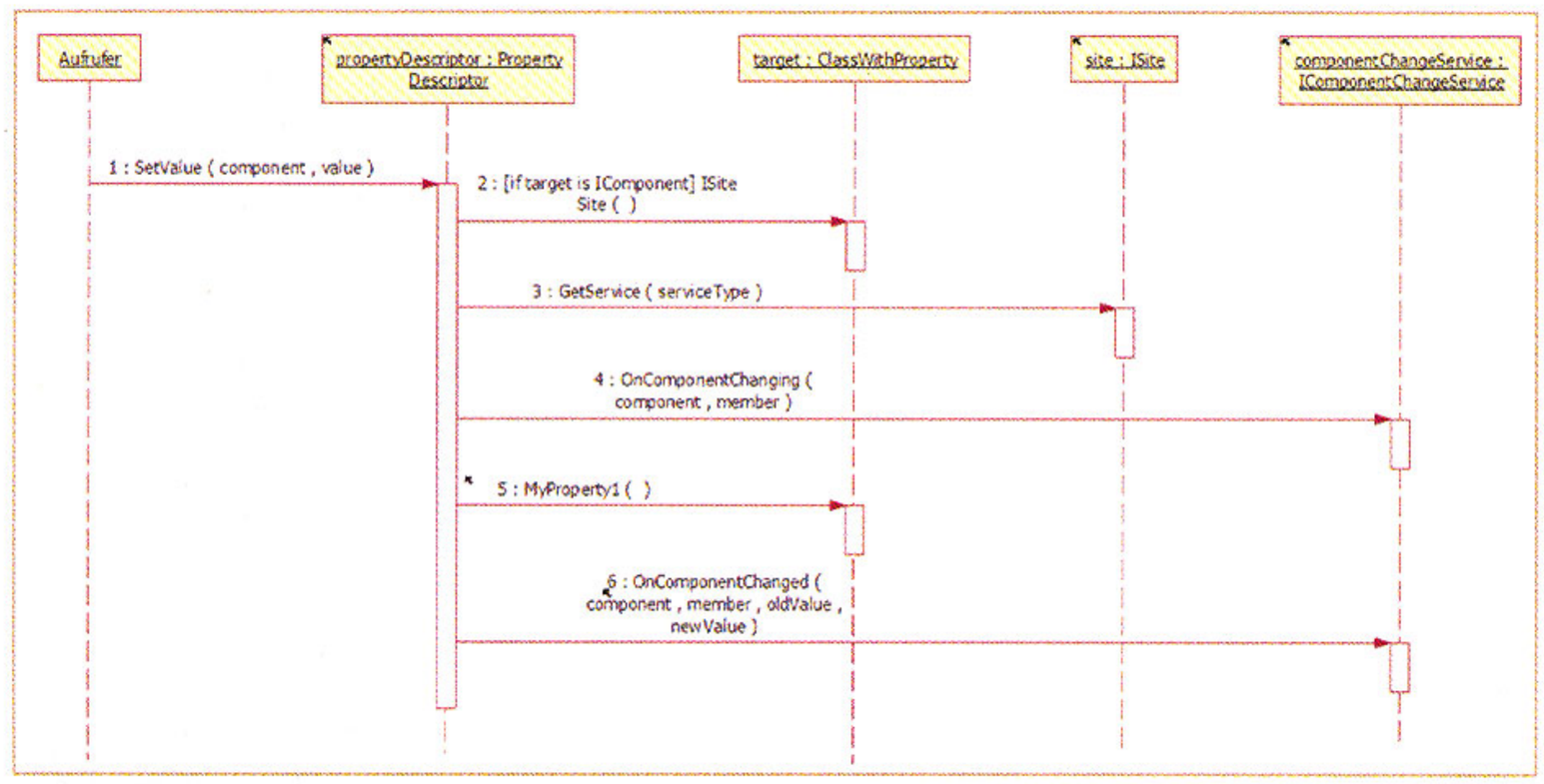

Abb. 2: Sequenzdiagramm der Abläufe beim Zugriff auf eine IComponent-Implementierung über einen PropertyDescriptor

pertyDescriptor beim Property-Zugriff kaum unterscheiden.

- Die Entitätsobjekte implementieren das Interface *ICustomTypeDescriptor* und geben in der Methode *ICustomTypeDescriptor.GetProperties(...)* nur Deskriptoren für Properties heraus, die auch tatsächlich in der Datenbank persistiert werden sollen.
- Für die von der Persistenzkomponente erwarteten technischen Properties wurden eigene Deskriptorklassen geschrieben, die in ihren Methoden *PropertyDescriptor.GetValue/PropertyDescriptor.SetValue* die für die Persistenzkomponente notwendige Funktionalität bereitstellen, ohne dass diese Properties tatsächlich an den Entitätsobjekten existieren. Diese Deskriptorklassen sind so allgemein gehalten, dass sie in allen Entitätsobjekten verwendet werden können.

Listing 8

```
namespace System
{
 interface IServiceProvider
 {
  // Gets the service object of the specified type.
  object GetService(Type type);
 }
}

namespace System.ComponentModel
{
 interface ISite : System.IServiceProvider
 {...}

 interface IComponent : System.IDisposable
 {
  ...
  /// Gets or sets the ISite associated with this
  /// IComponent.
  ISite Site { get ; set ;}
  ...
 }
}
```

Fazit: Komponententwickler tun gut daran, wo möglich Deskriptorobjekte statt Reflection einzusetzen. Und Komponentennutzer müssen mit Deskriptorobjekten und der Bedeutung des Interface *ICustomTypeDescriptor* vertraut sein.

Deskriptoren, Events und Attribute

An dieser Stelle zwei Hinweise, um die Beschreibung der Deskriptoren abzurunden:

1. Der Aufruf *TypeDescriptor.GetProperties(...)* kann nicht dahingehend parametrisiert werden, in welchem Kontext (z.B. von welcher Systemkomponente) er benutzt wird. Implementiert ein Zielobjekt das Interface *ICustomTypeDescriptor*, so gilt die eventuell vorgenommene Modifikation der Deskriptorobjekte für **alle** nutzenden Komponenten: Alle bekommen genau diejenigen Deskriptorobjekte, die der Aufruf von *ICustomTypeDescriptor.GetProperties(...)* liefert. Soll eine genauere Auswahlmöglichkeit geschaffen werden, welche Deskriptoren in welchem Kontext benutzt werden, müssen Metaattribute genutzt werden. Beim Aufruf von *TypeDescriptor.GetProperties(...)* kann angegeben werden, welche Metaattribute die gewünschten Properties haben müssen. Damit kann bei Bedarf gesteuert werden, welcher Deskriptor in welchem Kontext zum Einsatz kommt.
2. So wie mit der Klasse *PropertyDescriptor* ein Analogon für *PropertyInfo* existiert, gibt es vergleichbar zu *EventInfo* einen *EventDescriptor* zur Manipulation von Events. Alles hier über Properties gesagte gilt ähnlich auch für Events (Ausnahme: Für den mit der Methode *PropertyDescriptor.AddValueChanged(...)* verbundenen Benachrichtigungsmechanismus existiert in der Klasse *EventDescriptor* nichts Vergleichbares).

IComponent und ISite in der Theorie...

Mit den vorgestellten Fähigkeiten der Deskriptoren ist ihr Potenzial aber noch nicht erschöpft. Wer schon einmal nachgesehen hat, wie Windows-Forms-Controls untereinander kommunizieren, dem wird die auf den ersten Blick unnötig scheinende Benutzung von Deskriptoren auffallen: Obwohl die Properties oder Events, die ein Control an einem anderen nutzen will, bereits zur Compilezeit bekannt sind, werden statt direkter Aufrufe Deskriptoren verwendet. Wozu dieser Umstand? Und warum erschließt sich der Sinn des Umstands so schwer? Die zweite Frage ist leichter zu beantworten: Die im Folgenden beschriebene Funktionalität von *System.ComponentModel* ist unzureichend dokumentiert und teilweise nur durch Lesen des Codes der .NET-Klassen zu verstehen. Zur ersten Frage des „Wozu?“ fällt die Antwort komplizierter aus: Mithilfe der Deskriptoren wird Funktionalität umgesetzt, bei der alle Änderungen an Properties oder Events eines Controls zu einer Benachrichtigung anderer Objekte über diese anstehenden bzw. durchgeführten Modifikationen führen.

Dieser Benachrichtigungsprozess ist mehrstufig: Windows-Forms-Controls implementieren das Interface *IComponent*, das eine Sonderrolle im Zusammenhang mit Deskriptoren spielt. Bei einem Zugriff über Deskriptoren prüfen diese, ob das Zielobjekt des Zugriffs das Interface *IComponent* implementiert, das das Vorhandensein eines Property *Site* mit dem Rückgabetyp *ISite* fordert. *ISite* leitet sich vom Interface *System.IServiceProvider* ab. Die relevanten Teile der Interfaces *IComponent*, *ISite* und *IServiceProvider*

sehen wie in Listing 8 aus. Implementiert das Zielobjekt des Deskriptors das Interface *IComponent*, wird beim Zugriff durch den Deskriptor das Property *IComponent.Site* des Zielobjekts abgefragt. Ist die Site des Zielobjekts belegt, fragen die Deskriptoren weiter – sie fordern von der Site mithilfe der Methode *ISite.GetService(...)* ein Serviceobjekt mit dem Interface *IComponentChangeService* an (Listing 9).

Hat der Deskriptor ein Serviceobjekt mit dieser Schnittstelle bekommen, wird es von den Deskriptoren immer über anstehende und durchgeführte Änderungen einer Komponente informiert. Das Serviceobjekt kann Änderungen des Zielobjekts über Deskriptoren sogar verhindern, indem es in seiner Methode *IComponentChangeService.OnComponentChanging(...)* die Exception *CheckOutException.Cancel* wirft. In Kurzform: Implementiert ein Zielobjekt eines Deskriptors das Interface *IComponent* und ist die Site des Zielobjekts gesetzt und bietet diese ein Serviceobjekt mit geeigneter Schnittstelle, so wird dieses Serviceobjekt über anstehende und durchgeführte Änderungen des Zielobjekts informiert und kann Änderungen sogar blockieren.

IComponent und ISite in der Praxis

Aber kann der Mechanismus mit Site und Serviceobjekt auch nutzbringend zur Laufzeit einer Anwendung eingesetzt werden? Er kann – im bereits erwähnten Informationssystem *PRICE* manipulieren Anwendungsentwickler die GUI-Controls in der Regel nicht direkt, sondern nutzen eigens erstellte Hilfsmethoden und -objekte. Diese Helfer implementieren allgemeine Konzepte oder Teile der Anwendungsfachlichkeit. Wie in vielen anderen Systemen ist es auch in dieser Anwendung sinnvoll, dass zur Laufzeit eine zentrale Komponente der Benutzeroberfläche über alle Änderungen der Controls informiert wird. Diese zentrale Komponente kann ein zentrales Logging implementieren, sie kann beispielsweise prüfen, ob die beabsichtigten Modifikationen im augenblicklichen Gesamtzustand des Systems erlaubt sind.

Die automatische Benachrichtigung der zentralen Komponente lässt sich auf zwei Weisen erzielen: 1. Als erste Möglichkeit könnte den Hilfsmethoden und -objekten die zentrale zu informierende Komponente bekannt sein. Die Helfer würden alle Aktionen direkt an diese zentrale Komponente melden. Diese Variante hat aber Nachteile:

- Die zentrale Komponente müsste bereits zur Compilezeit überall im System bekannt und zugreifbar sein. Dies würde die Kopplung der Systembestandteile untereinander erhöhen, generelles Ziel ist aber das Gegenteil, nämlich deren Entkopplung.
- Hilfskonstrukte, die allgemeine und damit wieder verwendbare Konzepte implementieren, würden durch die Implementierung eines zusätzlichen Konzepts (Information einer zentralen Komponente) überfrachtet – Wartung und Wiederverwendbarkeit damit erschwert.

2. Sinnvoller ist die Umsetzung der zentralen Komponente mithilfe des Property: Allen Controls wird bei der Initialisierung der Anwendung eine *ISite*-Implementierung zugewiesen, die ein Serviceobjekt mit der Schnittstelle *IComponentChangeService* liefern kann. Nur diesem Serviceobjekt ist die zentrale GUI-Verwaltung bekannt. Die Hilfsmethoden und -objekte für den Control-Zugriff bedienen sich der Property- und Event-Deskriptoren. Diese informieren beim Zugriff auf die Controls das Serviceobjekt über alle Änderungen und dieses meldet relevante Daten an die zentrale GUI-Verwaltung. Damit muss die zentrale Komponente nur noch dem Serviceobjekt bekannt sein, Hilfsmethoden und -objekte wissen jetzt nichts mehr über die zentrale Komponente und sind damit besser wartbar und wieder verwendbar.

Fazit

Komponentenbasierte Software-Entwicklung ist auch im Kontext von .NET ein wichtiges Thema. Der Begriff der Komponente wird aber nicht eindeutig verwendet. Darum wurde der Begriff im ersten Teil der Artikelserie [1] definiert und motiviert, warum Komponenten so wichtig sind. Bedeutend ist die Trennung von Schnittstellen und deren Implementierung, was konkret belegt wurde. Wie man in einem Software-System auf die eingesetzten Komponenten zugreifen kann, wurde anhand von Beispielen im zweiten Teil [2] vorgestellt. Die Klassen des Namespace *System.ComponentModel* bieten viel Funktionalität, die zur Laufzeit einer Anwendung gewinnbringend eingesetzt werden kann – Funktionalität von unterschiedlicher Komplexität und leider auch von unterschiedlicher Güte der Dokumentation. In diesem Artikel wurden einige ausgewählte Aspekte dargestellt – von der Unterstützung bei der Datenkonvertierung über einen gegenüber Reflection reichhaltigeren Mechanismus für dynamischen Property- und Eventzugriff bis hin zu einem komplexeren Benachrichtigungsmechanismus bei der Modifikation von Komponenten. Leider ist weder die Klassendokumentation noch die Sekundärliteratur zu diesem Thema sehr ergiebig, sodass dem interessierten Entwickler zum Verständnis der Abläufe letztlich nur der Blick in den Code der .NET-Klassen bleibt. Es handelt sich aber um eine lohnende Lektüre und sowohl Entwickler wie auch Nutzer von Komponenten sollten mit den Konzepten und deren praktischer Umsetzung vertraut sein.

Listing 9

Die Definition von IComponentChangeService

```
namespace System.ComponentModel.Design
{
 interface IComponentChangeService
 {
  // Announces to the component that a particular
  // component is changing.
  void OnComponentChanging(object component,
                           MemberDescriptor member);

  // Announces to the component change service
  // that a particular component has changed.
  void OnComponentChanged(object component,
                          MemberDescriptor member,
   object oldValue, object newValue);
  ...
 }
}
```

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

Links & Literatur

[1] Alexander Ramisch + Burkhard Perkens-Golomb: .NET-Komponenten in großen Geschäftsanwendungen Teil 1, in: *dot.net magazin* 3.06

[2] Alexander Ramisch + Burkhard Perkens-Golomb: .NET im Einsatz Teil 2, in: *dot.net magazin* 5.06