Muster

WPF-Anwendungen mit dem Model-View-ViewModel-Entwurfsmuster

Josh Smith

Themen in diesem Artikel:

- Muster und WPF
- MVP-Muster
- Warum MVVM besser für WPF ist
- Erstellen einer Anwendung mit MVVM

In diesem Artikel werden folgende Technologien verwendet:
WPF, Datenbindung

Codedownload verfügbar in der MSDN-Codegalerie
Code online durchsuchen

Inhalt

Das Entwickeln der Benutzeroberfläche einer professionellen Softwareanwendung ist nicht einfach. Es kann eine schwer durchschaubare Mischung aus Daten, Interaktionsentwurf, visuellem Entwurf, Konnektivität, Multithreading, Sicherheit, Internationalisierung, Überprüfung, Komponententests und ein wenig Voodoo sein. In Anbetracht der Tatsache, dass eine Benutzeroberfläche das zugrunde liegende System verfügbar macht und die nicht voraussagbaren stilistischen Anforderungen ihrer Benutzer zufrieden stellen muss, kann dies der brisanteste Bereich vieler Anwendungen sein.

Es gibt beliebte Entwurfsmuster die hierbei helfen können, aber es kann schwierig sein, die vielfältigen Probleme zu trennen und in Angriff zu nehmen. Je komplizierter die Muster sind, desto wahrscheinlicher ist es, dass später Abkürzungen genommen werden, die alle vorherigen Bemühungen, es richtig zu machen, untergraben.

Dafür sind nicht immer die Entwurfsmuster verantwortlich. Manchmal verwenden wir komplizierte Entwurfsmuster, für die viel Code geschrieben werden muss, da sich die verwendete Benutzeroberflächenplattform nicht gut für ein einfacheres Muster eignet. Es ist eine Plattform erforderlich, mit der Benutzeroberflächen mithilfe einfacher, erprobter Entwurfsmuster, die von Entwicklern genehmigt wurden, leicht erstellt werden können. Glücklicherweise erfüllt Windows Presentation Foundation (WPF) genau diese Vorgabe.

Während die Softwarewelt in zunehmendem Maße WPF einsetzt, hat die WPF-Community ihr eigenes Ökosystem aus Mustern und Methoden entwickelt. In diesem Artikel stelle ich einige dieser bewährten Methoden zum Entwerfen und Implementieren von Clientanwendungen mit WPF vor. Durch die Nutzung einiger zentraler Features von WPF in Verbindung mit dem MVVM-Entwurfsmuster (Model-View-ViewModel) werde ich ein Beispielprogramm durcharbeiten, das zeigt, wie einfach es sein kann, eine WPF-Anwendung „richtig" zu erstellen.

Am Ende dieses Artikels wird klar sein, wie Datenvorlagen, Befehle, Datenbindung, das Ressourcensystem und das MVVM-Muster zusammenpassen, um ein einfaches, testbares, stabiles Framework zu erstellen, auf dem jede beliebige WPF-Anwendung gedeihen kann. Das Demoprogramm, das diesen Artikel begleitet, kann als Vorlage für eine echte WPF-Anwendung dienen, die MVVM als ihre Kernarchitektur verwendet. Die Komponententests in der Demolösung zeigen, wie einfach es ist, die Funktionalität der Benutzeroberfläche einer Anwendung zu testen, wenn diese Funktionalität in einem Satz von ViewModel-Klassen enthalten ist. Bevor es ins Detail geht, möchte ich darauf eingehen, warum Sie ein Muster wie MVVM überhaupt verwenden sollten.

Ordnung oder Chaos

Es ist unnötig und kontraproduktiv, Entwurfsmuster in einem einfachen Programm wie „Hello, World!" zu verwenden. Jeder kompetente Entwickler versteht ein paar Zeilen Code auf einen Blick. Bei einer zunehmenden Anzahl von Features in einem Programm nimmt jedoch die Anzahl von Codezeilen und „beweglichen Teilen" entsprechend zu. Schließlich bestärken die Komplexität eines Systems und die darin enthaltenen wiederkehrenden Probleme die Entwickler darin, ihren Code so zu organisieren, dass er leichter verständlich ist und einfacher diskutiert und erweitert werden kann und dass Fehler leichter behoben werden

MSDN Magazine Blog

Context-Aware Dialogue with Kinect: Part 2 of our Q&A with Leland Holmquest
Earlier this month I posted part of an interview with Leland Holmquest, who wrote our fascinating April issue exploration of the Kinect SDK (Context-A... More...
Montag, Apr 30

Script Junkie: Leveraging the jQuery UI Widget Factory with Project Silk
Things continue to hop over at Script Junkie. New to the site is Leveraging the jQuery UI Widget Factory with Project Silk, a feature article by Andre... More...
Freitag, Apr 13

More MSDN Magazine Blog entries >

Current Issue

Browse All MSDN Magazines

Subscribe to MSDN Flash newsletter

können. Wir verringern das kognitive Chaos eines komplexen Systems durch Anwenden bekannter Namen für bestimmte Entitäten im Quellcode. Wir legen den Namen, der auf einen Codeabschnitt angewendet werden soll, durch Betrachten seiner funktionalen Rolle im System fest.

Entwickler strukturieren ihren Code oft absichtlich gemäß einem Entwurfsmuster, statt zuzulassen, dass sich die Muster organisch entwickeln. Beide Ansätze sind in Ordnung, aber in diesem Artikel untersuche ich die Vorteile der expliziten Verwendung von MVVM als Architektur einer WPF-Anwendung. Die Namen bestimmter Klassen enthalten bekannte Begriffe aus dem MVVM-Muster, indem sie beispielsweise mit „ViewModel“ enden, wenn es sich bei der Klasse um die Abstraktion einer Ansicht handelt. Mit diesem Ansatz wird das bereits erwähnte kognitive Chaos vermieden. Stattdessen können Sie durchaus in einem Zustand des kontrollierten Chaos existieren, der in den meisten professionellen Softwareentwicklungsprojekten der Normalzustand ist!

Die Entwicklung von Model-View-ViewModel

Seitdem damit begonnen wurde, Softwarebenutzeroberflächen zu erstellen, gibt es beliebte Entwurfsmuster, die dabei helfen. Das MVP-Muster (Model-View-Presenter) beispielsweise ist auf verschiedenen Programmierplattformen für Benutzeroberflächen sehr beliebt. MVP ist eine Variante des Model-View-Controller-Musters, das es bereits seit Jahrzehnten gibt. Wenn Sie das MVP-Muster noch nie verwendet haben, finden Sie hier eine vereinfachte Erklärung. Auf dem Bildschirm sehen Sie die Ansicht (View). Die Daten, die dort angezeigt werden, sind das Modell, und der Presenter verknüpft die beiden. Die Ansicht verlässt sich auf einen Presenter, der sie mit Modelldaten füllt, auf Benutzereingaben reagiert, Eingabeüberprüfung bietet (vielleicht durch Delegieren an das Modell) und andere ähnliche Aufgaben durchführt. Wenn Sie mehr über MVP erfahren möchten, schlage ich vor, dass Sie den Artikel der Rubrik „Entwurfsmuster“ vom August 2006 von Jean-Paul Boodhoo lesen.

Im Jahr 2004 veröffentliche Martin Fowler einen Artikel über ein Muster namens Presentation Model (PM). Das PM-Muster ähnelt MVP, da es eine Ansicht von ihrem Verhalten und Zustand trennt. Der interessante Teil beim PM-Muster besteht darin, dass die Abstraktion einer Ansicht erstellt wird, die als Presentation Model bezeichnet wird. Die Ansicht wird dann lediglich zu einer Darstellung des Presentation Model. In seiner Erklärung zeigt Fowler, dass das Presentation Model seine Ansicht häufig aktualisiert, sodass beide miteinander synchronisiert sind. Diese Synchronisierungslogik ist als Code in den Presentation Model-Klassen vorhanden.

Im Jahr 2005 stellte John Gossman, derzeit einer der WPF- und Silverlight-Architekten bei Microsoft, das Model-View-ViewModel (MVVM)-Muster in seinem Blog vor. MVVM ist identisch mit dem Presentation Model von Fowler, da beide Muster eine Abstraktion einer Ansicht aufweisen, die den Zustand und das Verhalten einer Ansicht enthält. Fowler führte das Presentation Model als Mittel zum Erstellen der Abstraktion einer Ansicht ein, die von einer Benutzeroberflächenplattform unabhängig ist, während Gossman MVVM als standardisierte Möglichkeit einführte, die zentralen Features von WPF für das einfachere Erstellen von Benutzeroberflächen zu nutzen. In diesem Sinn betrachte ich MVVM als eine Spezialisierung des allgemeineren PM-Musters, das speziell an die WPF- und die Silverlight-Plattform angepasst ist.

In dem hervorragenden Artikel Prism: Muster für das Erstellen zusammengesetzter Anwendungen mit WPF von Glenn Block in der Ausgabe vom September 2008 wird die Microsoft Composite Application Guidance for WPF erläutert. Der Begriff „ViewModel“ wird dort nie verwendet. Stattdessen dient der Begriff „Presentation Model“ zum Beschreiben der Abstraktion einer Ansicht. In diesem Artikel wird jedoch das Muster als MVVM und die Abstraktion einer Ansicht als ViewModel bezeichnet. Ich habe festgestellt, dass diese Terminologie in der WPF- und Silverlight-Community weiter verbreitet ist.

Im Unterschied zum Presenter in MVP benötigt ein ViewModel keinen Verweis auf eine Ansicht. Die Ansicht wird an die Eigenschaften eines ViewModel gebunden, das wiederum Daten verfügbar macht, die in Modellobjekten und anderen für diese Ansicht spezifischen Zuständen enthalten sind. Das Erstellen der Bindungen zwischen Ansicht und ViewModel ist einfach, da ein ViewModel-Objekt als DataContext einer Ansicht festgelegt wird. Wenn sich Eigenschaftswerte im ViewModel ändern, werden diese neuen Werte automatisch über die Datenbindung an die Ansicht weitergeleitet. Wenn der Benutzer auf eine Schaltfläche in der Ansicht klickt, wird im ViewModel ein Befehl ausgeführt, um die angeforderte Aktion durchzuführen. Das ViewModel, aber niemals die Ansicht, führt alle an den Modelldaten vorgenommenen Änderungen durch.

Die Ansichtsklassen wissen nicht, dass die Modellklassen existieren, während dem ViewModel und dem Modell die Ansicht nicht bekannt ist. Das Modell weiß nichts darüber, dass das ViewModel und die Ansicht existieren. Es ist ein sehr lose gekoppelter Entwurf, der sich in vielerlei Hinsicht bezahlt macht, wie Sie bald sehen werden.

Warum WPF-Entwickler MVVM mögen

Wenn ein Entwickler erst einmal mit WPF und MVVM vertraut ist, kann es schwierig sein, zwischen beiden zu unterscheiden. MVVM ist die Verkehrssprache der WPF-Entwickler, da es gut zur WPF-Plattform passt, und WPF wurde so entworfen, dass es einfach ist, Anwendungen (unter anderem) mithilfe des MVVM-Musters zu erstellen. Tatsächlich hat Microsoft intern MVVM zum Entwickeln von WPF-Anwendungen, wie beispielsweise Microsoft Expression Blend, verwendet, während die zentrale WPF-Plattform entwickelt wurde. Viele Aspekte von WPF, z. B. das bescheiden wirkende Steuerungsmodell und die Datenvorlagen, nutzen die von MVVM geförderte starke Trennung zwischen Anzeige sowie Zustand und Verhalten.

Der wichtigste Aspekt von WPF, der MVVM zu einem hervorragenden Muster macht, ist die Datenbindungsinfrastruktur. Durch das Binden von Eigenschaften einer Ansicht an ein ViewModel kommt es zur losen Kopplung zwischen den beiden, wodurch in einem ViewModel kein Code geschrieben werden muss, der eine Ansicht direkt aktualisiert. Das Datenbindungssystem unterstützt auch die Eingabeüberprüfung, die eine standardisierte Möglichkeit zum Übertragen von Prüfungsfehlern an eine Ansicht bietet.

Zwei andere Features von WPF, die dieses Muster so nützlich machen, sind Datenvorlagen und das Ressourcensystem. Datenvorlagen wenden Ansichten auf ViewModel-Objekte an, die in der Benutzeroberfläche angezeigt werden. Sie können Vorlagen in XAML deklarieren, und das Ressourcensystem kann diese Vorlagen dann für Sie automatisch zur Laufzeit suchen und anwenden. Weitere Informationen zu Bindung und Datenvorlagen finden Sie in meinem Artikel Daten und WPF: Anpassen der Datenanzeige mit Datenbindung und WPF in der Ausgabe vom Juli 2008.

Wenn es in WPF nicht die Unterstützung für Befehle gäbe, wäre das MVVM-Muster viel weniger leistungsfähig. In diesem Artikel erfahren Sie, wie ein ViewModel Befehle für eine Ansicht verfügbar machen kann, sodass die Ansicht auf seine Funktionalität zugreifen kann. Wenn Sie mit der Befehlseingabe nicht vertraut sind, empfehle ich Ihnen den umfassenden Artikel WPF für Fortgeschrittene: Routingereignisse und -befehle in WPF von Brian Noyes, der in der Ausgabe vom September 2008 veröffentlicht wurde.

Neben den Features von WPF (und Silverlight 2), die MVVM zu einer natürlichen Methode zum Strukturieren

einer Anwendung machen, ist das Muster auch deshalb beliebt, weil an ViewModel-Klassen einfach Komponententests durchgeführt werden können. Wenn die Interaktionslogik einer Anwendung in einem Satz ViewModel-Klassen enthalten ist, können Sie problemlos Code schreiben, der sie testet. In gewisser Hinsicht sind Ansichten und Komponententests einfach zwei verschiedene Arten von ViewModel-Nutzern. Das Vorhandensein einer Testreihe für die ViewModels einer Anwendung bietet kostenlose und schnelle Regressionstests, wodurch die Kosten beim Verwalten einer Anwendung im Laufe der Zeit verringert werden.

Neben der Unterstützung für die Erstellung automatisierter Regressionstests kann die Testbarkeit von ViewModel-Klassen dazu beitragen, dass Benutzeroberflächen, deren visuelles Design einfach ist, richtig entworfen werden. Wenn Sie eine Anwendung entwerfen, können Sie oftmals entscheiden, ob etwas in der Ansicht oder im ViewModel enthalten sein sollte, indem Sie sich vorstellen, dass Sie einen Komponententest schreiben möchten, der das ViewModel nutzt. Wenn Sie Komponententests für das ViewModel schreiben können, ohne Benutzeroberflächenobjekte zu erstellen, können Sie das Design des ViewModel ebenfalls vollständig erstellen, da keine Abhängigkeiten von bestimmten visuellen Elementen vorhanden sind.

Schließlich können Entwickler, die mit visuellen Designern arbeiten, mithilfe von MVVM viel leichter einen reibungslosen Designer-/Entwicklerworkflow erstellen. Da eine Ansicht nur ein willkürlicher Nutzer eines ViewModel ist, ist es leicht, eine Ansicht einfach zu entfernen und eine neue Ansicht zum Rendern eines ViewModel einzufügen. Dieser einfache Schritt ermöglicht das schnelle Erstellen eines Prototyps und das Bewerten von Benutzeroberflächen, die von den Designern erstellt wurden.

Das Entwicklungsteam kann sich auf das Erstellen stabiler ViewModel-Klassen konzentrieren, während das Entwurfsteam schwerpunktmäßig benutzerfreundliche Ansichten erstellen kann. Für das Verknüpfen der Arbeitsergebnisse beider Teams ist möglicherweise wenig mehr nötig als sicherzustellen, dass in der XAML-Datei einer Ansicht die richtigen Bindungen vorhanden sind.

Die Demoanwendung

Bisher habe ich die Geschichte von MVVM und seine Verwendungstheorie vorgestellt. Zudem wurde untersucht, warum es unter WPF-Entwicklern so beliebt ist. Nun ist es an der Zeit, die Ärmel hochzukrempeln und das Muster in Aktion zu sehen. In der Demoanwendung, die diesen Artikel begleitet, wird MVVM auf vielfältige Weise verwendet. Sie bietet umfangreiche Beispiele, um die Konzepte in einen bedeutsamen Zusammenhang zu stellen. Ich habe die Demoanwendung in Visual Studio 2008 SP1 und mit Microsoft .NET Framework 3.5 SP1 erstellt. Die Komponententests werden im Komponententestsystem von Visual Studio ausgeführt.

Die Anwendung kann beliebig viele „Arbeitsbereiche“ enthalten, die der Benutzer jeweils durch Klicken auf einen Befehlslink im Navigationsbereich links öffnen kann. Alle Arbeitsbereiche sind in einem TabControl im Hauptinhaltsbereich vorhanden. Der Benutzer kann einen Arbeitsbereich durch Klicken auf die Schaltfläche „Close“ (Schließen) auf dem Registerkartenelement dieses Arbeitsbereichs schließen. Die Anwendung hat zwei verfügbare Arbeitsbereiche: „All Customers“ (Alle Kunden) und „New Customer“ (Neuer Kunde). Nach Ausführen der Anwendung und Öffnen einiger Arbeitsbereiche sieht die Benutzeroberfläche etwa wie in **Abbildung 1** aus.

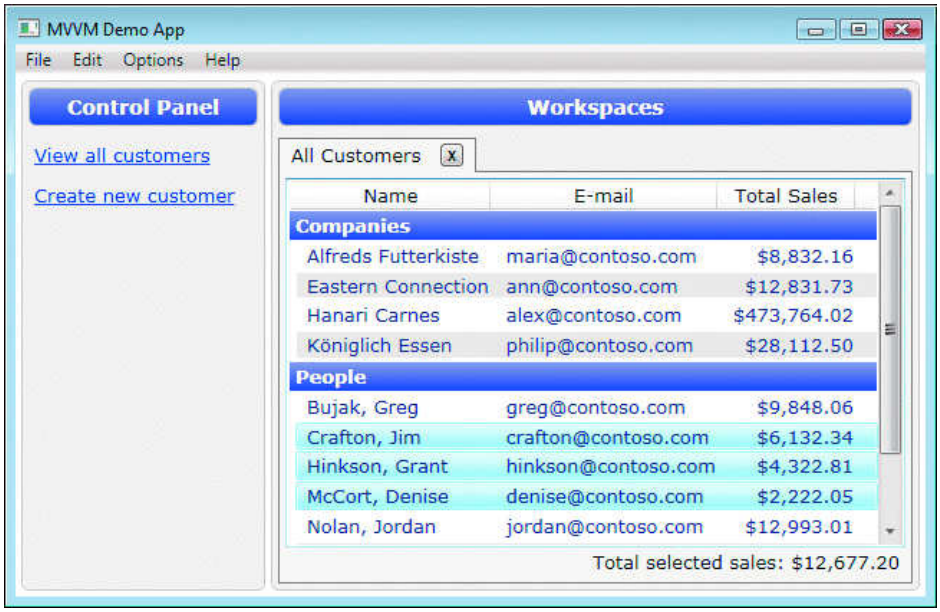

Abbildung 1 **Arbeitsbereiche**

Nur jeweils eine Instanz des Arbeitsbereichs „All Customers“ kann geöffnet sein, aber es können gleichzeitig beliebig viele Arbeitsbereiche „New Customer“ geöffnet sein. Wenn der Benutzer entscheidet, einen neuen Kunden zu erstellen, muss er das Dateneingabeformular in **Abbildung 2** ausfüllen.

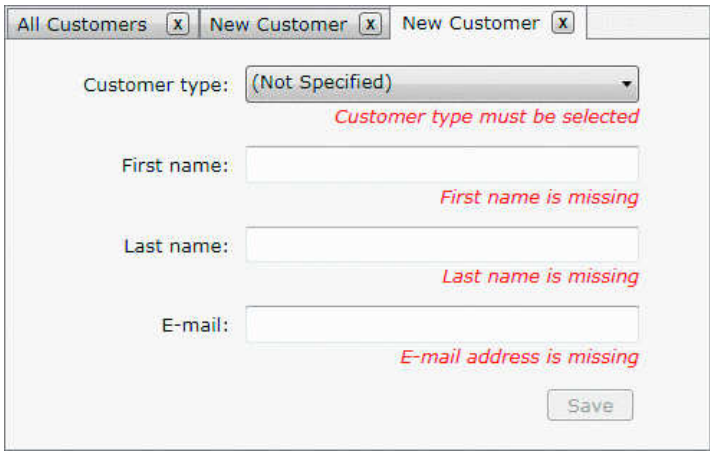

Abbildung 2 **Dateneingabeformular für neue Kunden**

Nach Ausfüllen des Dateneingabeformulars mit gültigen Werten und Klicken auf die Schaltfläche „Save" (Speichern) wird der Name des neuen Kunden auf der Registerkarte angezeigt, und der betreffende Kunde wird der Liste aller Kunden hinzugefügt. Die Anwendung unterstützt nicht das Löschen oder Bearbeiten eines vorhandenen Kunden, aber diese Funktionen und viele andere ähnliche Features können leicht implementiert werden, indem auf der vorhandenen Anwendungsarchitektur aufgebaut wird. Sie haben nun ein grundlegendes Verständnis der Demoanwendung, sodass wir uns ihrem Entwurf und ihrer Implementierung zuwenden können.

Übertragen der Befehlslogik

Jede Ansicht in der Anwendung hat eine leere CodeBehind-Datei. Ausgenommen sind die Standardcodebausteine, die InitializeComponent im Konstruktor der Klasse aufrufen. Im Grunde könnten Sie die CodeBehind-Dateien der Ansicht aus dem Projekt entfernen, und die Anwendung würde immer noch kompiliert und richtig ausgeführt werden. Trotz des Mangels an Ereignisbehandlungsmethoden in den Ansichten reagiert die Anwendung und erfüllt die Anforderungen der Benutzer, wenn auf Schaltflächen geklickt wird. Dies funktioniert aufgrund von Bindungen, die in der Command-Eigenschaft der Hyperlink-, Button- und MenuItem-Steuerelemente eingerichtet wurden, die in der Benutzeroberfläche angezeigt werden. Diese Bindungen stellen sicher, dass ICommand-Objekte, die vom ViewModel verfügbar gemacht werden, ausgeführt werden, wenn die Benutzer auf die Steuerelemente klicken. Sie können sich das Befehlsobjekt als einen Adapter vorstellen, mit dessen Hilfe die Funktionalität eines ViewModel leicht in einer in XAML deklarierten Ansicht genutzt werden kann.

Wenn ein ViewModel eine Instanzeigenschaft des Typs ICommand verfügbar macht, verwendet das Befehlsobjekt in der Regel dieses ViewModel-Objekt, um seine Aufgabe auszuführen. Ein mögliches Implementierungsmuster besteht darin, eine private geschachtelte Klasse innerhalb der ViewModel-Klasse zu erstellen, sodass der Befehl Zugriff auf private Member seines enthaltenden ViewModel hat und den Namespace nicht verunreinigt. Diese geschachtelte Klasse implementiert die ICommand-Schnittstelle, und ein Verweis auf das enthaltende ViewModel-Objekt wird in seinen Konstruktor injiziert. Das Erstellen einer geschachtelten Klasse, die ICommand für jeden Befehl implementiert, der von einem ViewModel verfügbar gemacht wird, kann jedoch die Größe der ViewModel-Klasse aufblähen. Mehr Code ist mit einem größeren Fehlerpotenzial verbunden.

In der Demoanwendung löst die RelayCommand-Klasse dieses Problem. RelayCommand ermöglicht Ihnen, die Logik des Befehls über Delegaten zu injizieren, die an seinen Konstruktor übergeben werden. Dieser Ansatz ermöglicht eine knappe, präzise Befehlsimplementierung in ViewModel-Klassen. RelayCommand ist eine vereinfachte Variante von DelegateCommand, das in der Microsoft Composite Application Library enthalten ist. Die RelayCommand-Klasse ist in **Abbildung 3** dargestellt.

Abbildung 3 Die RelayCommand-Klasse

```
public class RelayCommand : ICommand
{
    #region Fields

    readonly Action<object> _execute;
    readonly Predicate<object> _canExecute;

    #endregion // Fields

    #region Constructors

    public RelayCommand(Action<object> execute)
    : this(execute, null)
    {
    }

    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        if (execute == null)
            throw new ArgumentNullException("execute");

        _execute = execute;
        _canExecute = canExecute;
    }
    #endregion // Constructors

    #region ICommand Members
```

```
        [DebuggerStepThrough]
        public bool CanExecute(object parameter)
        {
            return _canExecute == null ? true : _canExecute(parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        public void Execute(object parameter)
        {
            _execute(parameter);
        }

        #endregion // ICommand Members
    }
```

Das CanExecuteChanged-Ereignis, das Teil der ICommand-Schnittstellenimplementierung ist, hat einige interessante Features. Es delegiert das Ereignisabonnement an das CommandManager.RequerySuggested-Ereignis. Dadurch wird sichergestellt, dass die WPF-Befehlsinfrastruktur jedes Mal, wenn sie die integrierten Befehle befragt, alle RelayCommand-Objekte fragt, ob sie ausgeführt werden können. Der folgende Code aus der CustomerViewModel-Klasse, die später eingehender untersucht wird, zeigt, wie ein RelayCommand mit Lambda-Ausdrücken konfiguriert wird:

```
RelayCommand _saveCommand;
public ICommand SaveCommand
{
    get
    {
        if (_saveCommand == null)
        {
            _saveCommand = new RelayCommand(param => this.Save(),
                param => this.CanSave );
        }
        return _saveCommand;
    }
}
```

ViewModel-Klassenhierarchie

Die meisten ViewModel-Klassen benötigen dieselben Features. Sie müssen oft die INotifyPropertyChanged-Schnittstelle implementieren, brauchen zumeist einen benutzerfreundlichen Anzeigenamen, und im Fall von Arbeitsbereichen muss die Möglichkeit vorhanden sein, dass sie geschlossen (das heißt, aus der Benutzeroberfläche entfernt) werden können. Dieses Problem eignet sich ganz natürlich für das Erstellen der einen oder anderen ViewModel-Basisklasse, sodass neue ViewModel-Klassen alle gebräuchlichen Funktionen von einer Basisklasse erben können. Die ViewModel-Klassen bilden die in **Abbildung 4** dargestellte Vererbungshierarchie.

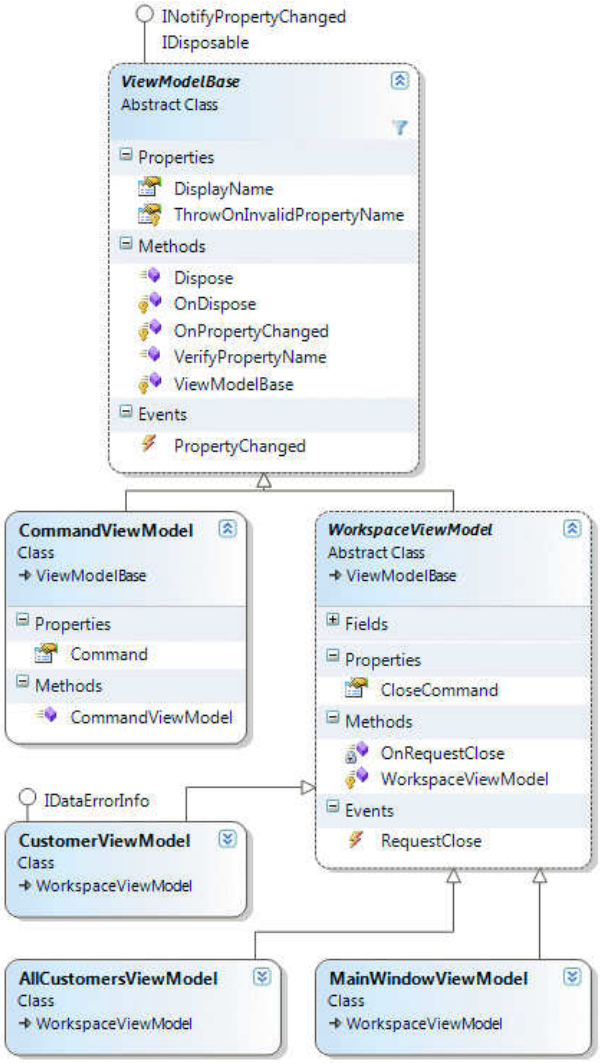

Abbildung 4 **Vererbungshierarchie**

Das Vorhandensein einer Basisklasse für alle Ihre ViewModels ist gar nicht erforderlich. Wenn Sie es bevorzugen, Features in Ihren Klassen durch Zusammenfassen vieler kleinerer Klassen bereitzustellen, statt Vererbung zu verwenden, ist dies kein Problem. Genau wie jedes andere Entwurfsmuster ist MVVM ein Satz von Richtlinien, nicht von Regeln.

ViewModelBase-Klasse

ViewModelBase ist die Basisklasse in der Hierarchie. Dies ist der Grund dafür, dass sie die häufig verwendete INotifyPropertyChanged-Schnittstelle implementiert und eine DisplayName-Eigenschaft hat. Die INotifyPropertyChanged-Schnittstelle enthält ein Ereignis namens „PropertyChanged“. Wenn eine Eigenschaft in einem ViewModel-Objekt einen neuen Wert hat, kann das PropertyChanged-Ereignis ausgelöst werden, um das WPF-Bindungssystem über den neuen Wert zu benachrichtigen. Nach Erhalt dieser Benachrichtigung fragt das Bindungssystem die Eigenschaft ab, und die gebundene Eigenschaft in einem Benutzeroberflächenelement erhält den neuen Wert.

Damit WPF weiß, welche Eigenschaft sich im ViewModel-Objekt geändert hat, macht die PropertyChangedEventArgs-Klasse eine PropertyName-Eigenschaft des Typs „String“ verfügbar. Sie müssen sorgfältig darauf achten, dass an dieses Ereignisargument der richtige Eigenschaftenname übergeben wird. Andernfalls fragt WPF die falsche Eigenschaft nach einem neuen Wert ab.

Ein interessanter Aspekt von ViewModelBase besteht darin, dass damit die Möglichkeit geboten wird zu überprüfen, ob eine Eigenschaft mit einem bestimmten Namen tatsächlich im ViewModel-Objekt vorhanden ist. Dies ist sehr nützlich bei der Umgestaltung, da beim Ändern des Namens einer Eigenschaft über das Visual Studio 2008-Umgestaltungsfeature Zeichenfolgen in Ihrem Quellcode, die den Namen dieser Eigenschaft enthalten, richtigerweise nicht aktualisiert werden. Das Auslösen des PropertyChanged-Ereignisses mit einem falschen Eigenschaftennamen im Ereignisargument kann zu Fehlern führen, die sich nur schwer aufspüren lassen, sodass dieses kleine Feature enorm viel Zeit sparen kann. Der Code von ViewModelBase, der diese nützliche Unterstützung hinzufügt, ist in **Abbildung 5** dargestellt.

Abbildung 5 Überprüfen einer Eigenschaft

```
// In ViewModelBase.cs
public event PropertyChangedEventHandler PropertyChanged;

protected virtual void OnPropertyChanged(string propertyName)
{
    this.VerifyPropertyName(propertyName);

    PropertyChangedEventHandler handler = this.PropertyChanged;
    if (handler != null)
```

```
        {
            var e = new PropertyChangedEventArgs(propertyName);
            handler(this, e);
        }
    }

    [Conditional("DEBUG")]
    [DebuggerStepThrough]
    public void VerifyPropertyName(string propertyName)
    {
        // Verify that the property name matches a real,
        // public, instance property on this object.
        if (TypeDescriptor.GetProperties(this)[propertyName] == null)
        {
            string msg = "Invalid property name: " + propertyName;

            if (this.ThrowOnInvalidPropertyName)
                throw new Exception(msg);
            else
                Debug.Fail(msg);
        }
    }
```

CommandViewModel-Klasse

Die einfachste konkrete ViewModelBase-Unterklasse ist CommandViewModel. Sie macht eine Eigenschaft namens „Command“ vom Typ „ICommand“ verfügbar. MainWindowViewModel macht eine Auflistung dieser Objekte durch ihre Commands-Eigenschaft verfügbar. Der Navigationsbereich auf der linken Seite des Hauptfensters zeigt einen Link für jedes CommandViewModel an, das von MainWindowViewModel verfügbar gemacht wird, wie beispielsweise „View all customers“ (Alle Kunden anzeigen) und „Create new customer“ (Neuen Kunden erstellen). Wenn der Benutzer auf einen Link klickt und damit einen dieser Befehle ausführt, wird im TabControl des Hauptfensters ein Arbeitsbereich geöffnet. Die CommandViewModel-Klassendefinition ist hier dargestellt:

```
public class CommandViewModel : ViewModelBase
{
    public CommandViewModel(string displayName, ICommand command)
    {
        if (command == null)
            throw new ArgumentNullException("command");

        base.DisplayName = displayName;
        this.Command = command;
    }

    public ICommand Command { get; private set; }
}
```

In der MainWindowResources.xaml-Datei ist ein DataTemplate vorhanden, dessen Schlüssel „CommandsTemplate“ lautet. MainWindow verwendet diese Vorlage, um die bereits erwähnte Auflistung von CommandViewModels zu rendern. Die Vorlage rendert einfach jedes CommandViewModel-Objekt als Link in einem ItemsControl-Element. Die Command-Eigenschaft jedes Hyperlinks ist an die Command-Eigenschaft eines CommandViewModel gebunden. Dieser XAML-Code ist in **Abbildung 6** dargestellt.

Abbildung 6 Rendern der Liste von Befehlen

```
<!-- In MainWindowResources.xaml -->
<!--
This template explains how to render the list of commands on
the left side in the main window (the 'Control Panel' area).
-->
<DataTemplate x:Key="CommandsTemplate">
  <ItemsControl ItemsSource="{Binding Path=Commands}">
    <ItemsControl.ItemTemplate>
      <DataTemplate>
        <TextBlock Margin="2,6">
          <Hyperlink Command="{Binding Path=Command}">
            <TextBlock Text="{Binding Path=DisplayName}" />
          </Hyperlink>
        </TextBlock>
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ItemsControl>
</DataTemplate>
```

MainWindowViewModel-Klasse

Wie Sie bereits im Klassendiagramm gesehen haben, leitet sich die WorkspaceViewModel-Klasse von ViewModelBase ab und fügt die Möglichkeit zum Schließen hinzu. Mit „Schließen“ meine ich, dass etwas zur Laufzeit den Arbeitsbereich aus der Benutzeroberfläche entfernt. Drei Klassen leiten sich von WorkspaceViewModel ab: MainWindowViewModel, AllCustomersViewModel und CustomerViewModel. Die Anforderung von MainWindowViewModel zum Schließen wird von der App-Klasse behandelt, die

MainWindow und sein ViewModel erstellt, wie in **Abbildung 7** dargestellt.

Abbildung 7 Erstellen von ViewModel

```
// In App.xaml.cs
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);

    MainWindow window = new MainWindow();

    // Create the ViewModel to which
    // the main window binds.
    string path = "Data/customers.xml";
    var viewModel = new MainWindowViewModel(path);

    // When the ViewModel asks to be closed,
    // close the window.
    viewModel.RequestClose += delegate
    {
        window.Close();
    };

    // Allow all controls in the window to
    // bind to the ViewModel by setting the
    // DataContext, which propagates down
    // the element tree.
    window.DataContext = viewModel;

    window.Show();
}
```

MainWindow enthält ein Menüelement, dessen Command-Eigenschaft an die CloseCommand-Eigenschaft von MainWindowViewModel gebunden ist. Wenn der Benutzer auf dieses Menüelement klickt, antwortet die App-Klasse folgendermaßen durch Aufrufen der Close-Methode des Fensters:

```
<!-- In MainWindow.xaml -->
<Menu>
  <MenuItem Header="_File">
    <MenuItem Header="_Exit" Command="{Binding Path=CloseCommand}" />
  </MenuItem>
  <MenuItem Header="_Edit" />
  <MenuItem Header="_Options" />
  <MenuItem Header="_Help" />
</Menu>
```

MainWindowViewModel enthält eine sichtbare Auflistung von WorkspaceViewModel-Objekten, die als Workspaces (Arbeitsbereiche) bezeichnet werden. Das Hauptfenster enthält ein TabControl, dessen ItemsSource-Eigenschaft an diese Auflistung gebunden ist. Jedes Registerkartenelement hat eine Schaltfläche „Close“ (Schließen), deren Command-Eigenschaft an das CloseCommand seiner entsprechenden WorkspaceViewModel-Instanz gebunden ist. Eine verkürzte Version der Vorlage, die jedes Registerkartenelement konfiguriert, ist im folgenden Code dargestellt. Der Code ist in MainWindowResources.xaml enthalten, und die Vorlage erklärt, wie ein Registerkartenelement mit einer Close-Schaltfläche gerendert wird.

```
<DataTemplate x:Key="ClosableTabItemTemplate">
  <DockPanel Width="120">
    <Button
      Command="{Binding Path=CloseCommand}"
      Content="X"
      DockPanel.Dock="Right"
      Width="16" Height="16"
      />
    <ContentPresenter Content="{Binding Path=DisplayName}" />
  </DockPanel>
</DataTemplate>
```

Wenn der Benutzer in einem Registerkartenelement auf die Schaltfläche „Close“ klickt, wird das CloseCommand dieses WorkspaceViewModel ausgeführt, sodass sein RequestClose-Ereignis ausgelöst wird. MainWindowViewModel überwacht das RequestClose-Ereignis seiner Arbeitsbereiche und entfernt auf Anforderung den Arbeitsbereich aus der Workspaces-Auflistung. Da die ItemsSource-Eigenschaft des TabControl von MainWindow an die sichtbare Auflistung von WorkspaceViewModels gebunden ist, führt das Entfernen eines Elements aus der Auflistung dazu, dass der entsprechende Arbeitsbereich aus dem TabControl entfernt wird. Diese Logik aus MainWindowViewModel ist in **Abbildung 8** dargestellt.

Abbildung 8 Entfernen eines Arbeitsbereichs aus der Benutzeroberfläche

```
// In MainWindowViewModel.cs

ObservableCollection<WorkspaceViewModel> _workspaces;
```

```
public ObservableCollection<WorkspaceViewModel> Workspaces
{
    get
    {
        if (_workspaces == null)
        {
            _workspaces = new ObservableCollection<WorkspaceViewModel>();
            _workspaces.CollectionChanged += this.OnWorkspacesChanged;
        }
        return _workspaces;
    }
}

void OnWorkspacesChanged(object sender, NotifyCollectionChangedEventArgs e)
{
    if (e.NewItems != null && e.NewItems.Count != 0)
        foreach (WorkspaceViewModel workspace in e.NewItems)
            workspace.RequestClose += this.OnWorkspaceRequestClose;

    if (e.OldItems != null && e.OldItems.Count != 0)
        foreach (WorkspaceViewModel workspace in e.OldItems)
            workspace.RequestClose -= this.OnWorkspaceRequestClose;
}

void OnWorkspaceRequestClose(object sender, EventArgs e)
{
    this.Workspaces.Remove(sender as WorkspaceViewModel);
}
```

Im UnitTests-Projekt enthält die MainWindowViewModelTests.cs-Datei eine Testmethode, die überprüft, ob diese Funktionalität ordnungsgemäß arbeitet. Die Leichtigkeit, mit der Sie Komponententests für ViewModel-Klassen erstellen können, ist ein wichtiger Vorteil des MVVM-Musters, weil dadurch einfaches Testen der Anwendungsfunktionalität ermöglicht wird, ohne Code schreiben zu müssen, der die Benutzeroberfläche verwendet. Diese Testmethode ist in **Abbildung 9** dargestellt.

Abbildung 9 Die Testmethode

```
// In MainWindowViewModelTests.cs
[TestMethod]
public void TestCloseAllCustomersWorkspace()
{
    // Create the MainWindowViewModel, but not the MainWindow.
    MainWindowViewModel target =
        new MainWindowViewModel(Constants.CUSTOMER_DATA_FILE);

    Assert.AreEqual(0, target.Workspaces.Count, "Workspaces isn't empty.");

    // Find the command that opens the "All Customers" workspace.
    CommandViewModel commandVM =
        target.Commands.First(cvm => cvm.DisplayName == "View all customers");

    // Open the "All Customers" workspace.
    commandVM.Command.Execute(null);
    Assert.AreEqual(1, target.Workspaces.Count, "Did not create viewmodel.");

    // Ensure the correct type of workspace was created.
    var allCustomersVM = target.Workspaces[0] as AllCustomersViewModel;
    Assert.IsNotNull(allCustomersVM, "Wrong viewmodel type created.");

    // Tell the "All Customers" workspace to close.
    allCustomersVM.CloseCommand.Execute(null);
    Assert.AreEqual(0, target.Workspaces.Count, "Did not close viewmodel.");
}
```

Anwenden einer Ansicht auf ein ViewModel

MainWindowViewModel fügt dem TabControl des Hauptfensters indirekt WorkspaceViewModel-Objekte hinzu und entfernt diese. Aufgrund der Verwendung von Datenbindung erhält die Content-Eigenschaft eines TabItem ein von ViewModelBase abgeleitetes Objekt zum Anzeigen. ViewModelBase ist kein Benutzeroberflächenelement, sodass es nicht über inhärente Unterstützung verfügt, um sich selbst zu rendern. Standardmäßig wird ein nicht visuelles Objekt in WPF durch Anzeigen der Ergebnisse eines Aufrufs seiner ToString-Methode in einem TextBlock gerendert. Dies ist eindeutig nicht das, was Sie brauchen, es sei denn, Ihre Benutzer möchten unbedingt den Typnamen unserer ViewModel-Klassen sehen!

Sie können WPF problemlos mitteilen, wie ein ViewModel-Objekt mithilfe typisierter DataTemplates gerendert wird. Einem typisierten DataTemplate ist kein x:Key-Wert zugewiesen, aber seine DataType-Eigenschaft ist auf eine Instanz der Type-Klasse gesetzt. Wenn WPF versucht, eines Ihrer ViewModel-Objekte zu rendern, prüft es, ob das Ressourcensystem ein typisiertes DataTemplate im Bereich hat, dessen DataType identisch mit dem Typ Ihres ViewModel-Objekts (oder eine Basisklasse davon) ist. Wenn es eine Vorlage findet, verwendet es diese zum Rendern des ViewModel-Objekts, auf das durch die Content-Eigenschaft des Registerkartenelements verwiesen wird.

Die MainWindowResources.xaml-Datei hat ein ResourceDicitionary. Dieses Wörterbuch wird der Ressourcenhierarchie des Hauptfensters hinzugefügt, sodass sich die darin enthaltenen Ressourcen im Ressourcenbereich des Fensters befinden. Wenn der Inhalt eines Registerkartenelements auf ein ViewModel-Objekt gesetzt ist, stellt ein typisiertes DataTemplate aus diesem Wörterbuch eine Ansicht (das heißt ein

Benutzersteuerelement) bereit, um es zu rendern, wie in **Abbildung 10** dargestellt.

Abbildung 10 Bereitstellen einer Ansicht

```
<!--
This resource dictionary is used by the MainWindow.
-->
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:vm="clr-namespace:DemoApp.ViewModel"
  xmlns:vw="clr-namespace:DemoApp.View"
  >

  <!--
  This template applies an AllCustomersView to an instance
  of the AllCustomersViewModel class shown in the main window.
  -->
  <DataTemplate DataType="{x:Type vm:AllCustomersViewModel}">
    <vw:AllCustomersView />
  </DataTemplate>

  <!--
  This template applies a CustomerView to an instance
  of the CustomerViewModel class shown in the main window.
  -->
  <DataTemplate DataType="{x:Type vm:CustomerViewModel}">
    <vw:CustomerView />
  </DataTemplate>

 <!-- Other resources omitted for clarity... -->

</ResourceDictionary>
```

Sie müssen keinen Code schreiben, der festlegt, welche Ansicht für ein ViewModel-Objekt angezeigt werden soll. Das WPF-Ressourcensystem nimmt Ihnen die schwere Arbeit ab, sodass Sie sich auf wichtigere Dinge konzentrieren können. In komplexeren Szenarios ist es möglich, die Ansicht programmgesteuert auszuwählen, aber in den meisten Situationen ist dies nicht erforderlich.

Datenmodell und Repository

Sie wissen nun, wie ViewModel/Objekte geladen, angezeigt und durch die Anwendungsshell geschlossen werden. Die allgemeine Grundstruktur ist jetzt vorhanden, und Sie können sich den Implementierungsinformationen zuwenden, die für die Domäne der Anwendung spezifischer sind. Bevor wir uns intensiv mit den zwei Arbeitsbereichen der Anwendung („All Customers“ und „New Customer“) befassen, untersuchen wir zuerst das Datenmodell und die Datenzugriffsklassen. Der Entwurf dieser Klassen hat fast nichts mit dem MVVM-Muster zu tun, da Sie eine ViewModel-Klasse erstellen können, um fast jedes Datenobjekt so anzupassen, dass es WPF-freundlich ist.

Die einzige Modellklasse im Demoprogramm ist Customer. Diese Klasse hat eine Reihe von Eigenschaften, die Informationen zu einem Kunden eines Unternehmens repräsentieren, z. B. Vorname, Nachname und E-Mail-Adresse. Es bietet Überprüfungsmeldungen durch Implementieren der standardmäßigen IDataErrorInfo-Schnittstelle, die es bereits etliche Jahre gab, bevor WPF veröffentlicht wurde. In der Customer-Klasse ist nichts vorhanden, das nahelegt, dass sie in einer MVVM-Architektur oder in einer WPF-Anwendung verwendet wird. Die Klasse könnte genauso gut aus einer älteren Unternehmensbibliothek stammen.

Daten müssen irgendwoher kommen und sich irgendwo befinden. In dieser Anwendung lädt und speichert eine Instanz der CustomerRepository-Klasse alle Customer-Objekte. Sie lädt die Kundendaten aus einer XML-Datei, aber der Typ der externen Datenquelle ist irrelevant. Die Daten könnten aus einer Datenbank, von einem Webdienst, aus einer Named Pipe, einer Datei auf einem Datenträger oder sogar von Brieftauben stammen – es ist völlig egal. Solange Sie ein .NET-Objekt mit einigen Daten haben, kann das MVVM-Muster diese Daten unabhängig von ihrer Herkunft auf dem Bildschirm darstellen.

Die CustomerRepository-Klasse macht einige Methoden verfügbar, die Ihnen ermöglichen, alle verfügbaren Customer-Objekte abzurufen, ein neues Customer-Objekt dem Repository hinzuzufügen und zu überprüfen, ob ein Customer-Objekt bereits im Repository vorhanden ist. Da die Anwendung dem Benutzer nicht ermöglicht, einen Kunden zu löschen, ermöglicht das Repository das Entfernen eines Kunden ebenfalls nicht. Das CustomerAdded-Ereignis wird ausgelöst, wenn ein neuer Kunde über die AddCustomer-Methode in das CustomerRepository aufgenommen wird.

Das Datenmodell dieser Anwendung ist natürlich sehr klein im Vergleich dazu, was für echte Geschäftsanwendungen erforderlich ist, aber das ist unwichtig. Es ist wichtig zu verstehen, wie die ViewModel-Klassen Customer und CustomerRepository verwenden. Beachten Sie, dass CustomerViewModel ein Wrapper für das Customer-Objekt ist. Es macht den Zustand eines Customer-Objekts und andere vom CustomerView-Steuerelement verwendete Zustände über einen Satz von Eigenschaften verfügbar. CustomerViewModel dupliziert nicht den Zustand eines Customer-Objekts. Es macht ihn einfach über Delegierung verfügbar, und zwar folgendermaßen:

```
public string FirstName
{
    get { return _customer.FirstName; }
    set
    {
        if (value == _customer.FirstName)
            return;
        _customer.FirstName = value;
        base.OnPropertyChanged("FirstName");
    }
```

```
}
```

Wenn der Benutzer einen neuen Kunden erstellt und im CustomerView-Steuerelement auf die Schaltfläche „Save“ klickt, wird das dieser Ansicht zugeordnete CustomerViewModel das neue Customer-Objekt zu CustomerRepository hinzufügen. Dies führt dazu, dass das CustomerAdded-Ereignis des Repository ausgelöst wird, das AllCustomersViewModel mitteilt, dass es seiner AllCustomers-Auflistung ein neues CustomerViewModel hinzufügen sollte. CustomerRepository agiert gewissermaßen als Synchronisierungsmechanismus zwischen verschiedenen ViewModels, die mit Customer-Objekten zu tun haben. Sie können sich dies etwa wie das Verwenden des Mediator-Entwurfsmusters vorstellen. In den nächsten Abschnitten werde ich näher darauf eingehen, wie dies funktioniert, aber zuerst sollten Sie sich das Diagramm in **Abbildung 11** ansehen, um ungefähr zu verstehen, wie sich die verschiedenen Teile zusammenfügen.

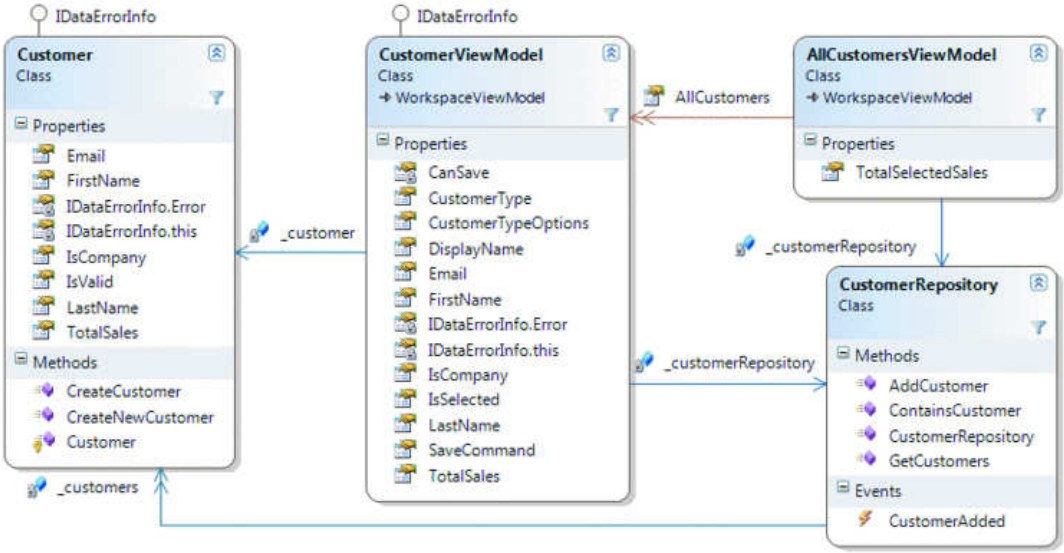

Abbildung 11 **Kundenbeziehungen**

Dateneingabeformular für neue Kunden

Wenn der Benutzer auf den Link „Create new customer“ (Neuen Kunden erstellen) klickt, fügt MainWindowViewModel seiner Liste von Arbeitsbereichen ein neues CustomerViewModel hinzu, das von einem CustomerView-Steuerelement angezeigt wird. Nachdem der Benutzer gültige Werte in die Eingabefelder eingegeben hat, wird die Schaltfläche „Save“ aktiviert, sodass der Benutzer die neuen Kundeninformationen beibehalten kann. Es handelt sich um ein ganz gewöhnliches, normales Dateneingabeformular mit Eingabeüberprüfung und einer Schaltfläche zum Speichern.

Die Customer-Klasse verfügt über integrierte Überprüfungsunterstützung, die über die Implementierung ihrer IDataErrorInfo-Schnittstelle verfügbar ist. Diese Überprüfung stellt sicher, dass der Kunde einen Vornamen, eine wohlgeformte E-Mail-Adresse und einen Nachnamen hat, wenn es sich bei ihm um eine Person handelt. Wenn die IsCompany-Eigenschaft von Customer den Wert „true“ zurückgibt, kann die LastName-Eigenschaft keinen Wert haben (da ein Unternehmen üblicherweise keinen Nachnamen hat). Diese Überprüfungslogik mag aus der Perspektive des Customer-Objekts sinnvoll scheinen, erfüllt aber nicht die Anforderungen der Benutzeroberfläche. Die Benutzeroberfläche erfordert, dass der Benutzer auswählt, ob ein neuer Kunde eine Person oder ein Unternehmen ist. Die Customer Type-Auswahl hat anfänglich den Wert „(Not Specified)“. Wie kann die Benutzeroberfläche dem Benutzer mitteilen, dass der Kundentyp nicht angegeben ist, wenn die IsCompany-Eigenschaft eines Kunden nur den Wert „true“ oder „false“ zulässt?

Wenn Sie vollständige Kontrolle über das gesamte Softwaresystem haben, könnten Sie die IsCompany-Eigenschaft so ändern, dass sie den Typ Nuallable<bool> hat, wodurch der Wert „unselected“ zugelassen würde. In der Praxis ist es jedoch nicht immer so einfach. Angenommen, Sie können die Customer-Klasse nicht ändern, da sie aus einer Legacybibliothek stammt, die einem anderen Team in Ihrem Unternehmen gehört. Was geschieht, wenn es aufgrund des vorhandenen Datenbankschemas keine einfache Möglichkeit gibt, den Wert „unselected“ beizubehalten? Was geschieht, wenn andere Anwendungen bereits die Customer-Klasse verwenden und sich darauf verlassen, dass die Eigenschaft ein normaler boolescher Wert ist? Auch hier ist ein ViewModel die Rettung.

Die Testmethode in **Abbildung 12** zeigt, wie diese Funktionalität in CustomerViewModel funktioniert. CustomerViewModel macht eine CustomerTypeOptions-Eigenschaft verfügbar, sodass die Customer Type-Auswahl drei Zeichenfolgen hat, die angezeigt werden. Es macht auch eine CustomerType-Eigenschaft verfügbar, die das ausgewählte String-Element in der Auswahl speichert. Wenn CustomerType festgelegt ist, ordnet es einem booleschen Wert den String-Wert für die IsCompany-Eigenschaft des zugrunde liegenden Customer-Objekts zu. **Abbildung 13** zeigt die beiden Eigenschaften.

Abbildung 12 Die Testmethode

```
// In CustomerViewModelTests.cs
[TestMethod]
public void TestCustomerType()
{
    Customer cust = Customer.CreateNewCustomer();
    CustomerRepository repos = new CustomerRepository(
        Constants.CUSTOMER_DATA_FILE);
    CustomerViewModel target = new CustomerViewModel(cust, repos);

    target.CustomerType = "Company"
    Assert.IsTrue(cust.IsCompany, "Should be a company");
```

```
    target.CustomerType = "Person";
    Assert.IsFalse(cust.IsCompany, "Should be a person");

    target.CustomerType = "(Not Specified)";
    string error = (target as IDataErrorInfo)["CustomerType"];
    Assert.IsFalse(String.IsNullOrEmpty(error), "Error message should
        be returned");
}
```

Abbildung 13 CustomerType-Eigenschaften

```
// In CustomerViewModel.cs

public string[] CustomerTypeOptions
{
    get
    {
        if (_customerTypeOptions == null)
        {
            _customerTypeOptions = new string[]
            {
                "(Not Specified)",
                "Person",
                "Company"
            };
        }
        return _customerTypeOptions;
    }
}
public string CustomerType
{
    get { return _customerType; }
    set
    {
        if (value == _customerType ||
            String.IsNullOrEmpty(value))
            return;

        _customerType = value;

        if (_customerType == "Company")
        {
            _customer.IsCompany = true;
        }
        else if (_customerType == "Person")
        {
            _customer.IsCompany = false;
        }

        base.OnPropertyChanged("CustomerType");
        base.OnPropertyChanged("LastName");
    }
}
```

Das CustomerView-Steuerelement enthält eine ComboBox, die an diese Eigenschaften gebunden ist, wie hier dargestellt:

```
<ComboBox
  ItemsSource="{Binding CustomerTypeOptions}"
  SelectedItem="{Binding CustomerType, ValidatesOnDataErrors=True}"
  />
```

Wenn sich das ausgewählte Element in dieser ComboBox ändert, wird die IDataErrorInfo-Schnittstelle der Datenquelle abgefragt, um zu sehen, ob der neue Wert gültig ist. Dies geschieht, weil bei der SelectedItem-Eigenschaftsbindung ValidatesOnDataErrors auf „true" gesetzt ist. Da die Datenquelle ein CustomerViewModel-Objekt ist, fragt das Bindungssystem dieses CustomerViewModel nach einem Überprüfungsfehler in der CustomerType-Eigenschaft. Meistens delegiert CustomerViewModel alle Anforderungen für Überprüfungsfehler an das Customer-Objekt, das es enthält. Da Customer aber keinen deaktivierten Zustand für die IsCompany-Eigenschaft kennt, muss die CustomerViewModel-Klasse die Überprüfung des neuen ausgewählten Elements im ComboBox-Steuerelement behandeln. Dieser Code ist in **Abbildung 14** dargestellt.

Abbildung 14 Überprüfen eines CustomerViewModel-Objekts

```
// In CustomerViewModel.cs
string IDataErrorInfo.this[string propertyName]
{
    get
    {
        string error = null;
```

```
        if (propertyName == "CustomerType")
        {
            // The IsCompany property of the Customer class
            // is Boolean, so it has no concept of being in
            // an "unselected" state. The CustomerViewModel
            // class handles this mapping and validation.
            error = this.ValidateCustomerType();
        }
        else
        {
            error = (_customer as IDataErrorInfo)[propertyName];
        }

        // Dirty the commands registered with CommandManager,
        // such as our Save command, so that they are queried
        // to see if they can execute now.
        CommandManager.InvalidateRequerySuggested();

        return error;
    }
}

string ValidateCustomerType()
{
    if (this.CustomerType == "Company" ||
       this.CustomerType == "Person")
        return null;

    return "Customer type must be selected";
}
```

Der Hauptaspekt dieses Codes besteht darin, dass die Implementierung von IDataErrorInfo durch CustomerViewModel Anforderungen für die ViewModel-spezifische Eigenschaftsüberprüfung behandeln und die anderen Anforderungen an das Customer-Objekt delegieren kann. Dies ermöglicht die Verwendung von Überprüfungslogik in Model-Klassen. Zudem steht zusätzliche Überprüfung von Eigenschaften zur Verfügung, die nur für ViewModel-Klassen Sinn ergeben.

Die Möglichkeit, ein CustomerViewModel zu speichern, steht in einer Ansicht über die SaveCommand-Eigenschaft zur Verfügung. Dieser Befehl verwendet die RelayCommand-Klasse, die bereits untersucht wurde, damit CustomerViewModel entscheiden kann, ob es sich selbst speichern kann und was zu tun ist, wenn es aufgefordert wird, seinen Zustand zu speichern. In dieser Anwendung bedeutet das Speichern eines neuen Kunden einfach, ihn einem CustomerRepository hinzuzufügen. Für die Entscheidung, ob der neue Kunde gespeichert werden kann, ist die Zustimmung von zwei Parteien erforderlich. Das Customer-Objekt muss gefragt werden, ob es gültig ist oder nicht, und das CustomerViewModel muss entscheiden, ob es gültig ist. Diese zweiteilige Entscheidung ist aufgrund der ViewModel-spezifischen Eigenschaften und der Überprüfung, die bereits untersucht wurde, notwendig. Die Speicherlogik für CustomerViewModel ist in **Abbildung 15** dargestellt.

Abbildung 15 Die Speicherlogik für CustomerViewModel

```
// In CustomerViewModel.cs
public ICommand SaveCommand
{
    get
    {
        if (_saveCommand == null)
        {
            _saveCommand = new RelayCommand(
                param => this.Save(),
                param => this.CanSave
                );
        }
        return _saveCommand;
    }
}

public void Save()
{
    if (!_customer.IsValid)
        throw new InvalidOperationException("...");

    if (this.IsNewCustomer)
        _customerRepository.AddCustomer(_customer);

    base.OnPropertyChanged("DisplayName");
}

bool IsNewCustomer
{
    get
    {
        return !_customerRepository.ContainsCustomer(_customer);
    }
}

bool CanSave
{
```

```
        get
        {
            return
                String.IsNullOrEmpty(this.ValidateCustomerType()) &&
                _customer.IsValid;
        }
    }
```

Die Verwendung eines ViewModel erleichtert hier sehr das Erstellen einer Ansicht, die ein Customer-Objekt anzeigen kann und Dinge wie den unselected-Zustand einer booleschen Eigenschaft zulässt. Es bietet auch die Möglichkeit, dem Kunden problemlos mitzuteilen, dass er seinen Zustand speichern soll. Wenn die Ansicht direkt an ein Customer-Objekt gebunden wäre, wäre für die Ansicht viel Code erforderlich, damit dies richtig funktioniert. In einer gut entworfenen MVVM-Architektur sollte das CodeBehind für die meisten Ansichten leer sein oder allenfalls nur Code enthalten, der die in dieser Ansicht enthaltenen Steuerelemente und Ressourcen bearbeitet. Manchmal ist es auch notwendig, im CodeBehind einer Ansicht Code zu schreiben, der mit einem ViewModel-Objekt interagiert, z. B. durch Einbinden eines Ereignisses oder Aufrufen einer Methode, die andernfalls nur sehr schwer aus dem ViewModel aufgerufen werden könnte.

Ansicht aller Kunden

Die Demoanwendung enthält auch einen Arbeitsbereich, der alle Kunden in einem ListView anzeigt. Die Kunden in der Liste sind danach gruppiert, ob sie ein Unternehmen oder eine Person sind. Der Benutzer kann einen oder mehrere Kunden auf einmal auswählen und sich die Summe ihres Gesamtumsatzes in der rechten unteren Ecke anzeigen lassen.

Die Benutzeroberfläche ist das AllCustomersView-Steuerelement, das ein AllCustomersViewModel-Objekt rendert. Jedes ListViewItem repräsentiert ein CustomerViewModel-Objekt in der AllCustomers-Auflistung, die vom AllCustomerViewModel-Objekt verfügbar gemacht wird. Im vorherigen Abschnitt haben Sie gesehen, wie ein CustomerViewModel als Dateneingabeformular gerendert werden kann, und jetzt wird genau dasselbe CustomerViewModel-Objekt als Element in einem ListView gerendert. Die CustomerViewModel-Klasse weiß nicht, von welchen visuellen Elementen sie angezeigt wird. Dies ist der Grund, warum diese Wiederverwendung möglich ist.

AllCustomersView erstellt die in ListView dargestellten Gruppen. Sie erreicht dies durch Binden von ItemsSource von ListView an ein CollectionViewSource, das wie in **Abbildung 16** konfiguriert ist.

Abbildung 16 CollectionViewSource

```
<!-- In AllCustomersView.xaml -->
<CollectionViewSource
  x:Key="CustomerGroups"
  Source="{Binding Path=AllCustomers}"
  >
  <CollectionViewSource.GroupDescriptions>
    <PropertyGroupDescription PropertyName="IsCompany" />
  </CollectionViewSource.GroupDescriptions>
  <CollectionViewSource.SortDescriptions>
    <!--
    Sort descending by IsCompany so that the ' True' values appear first,
    which means that companies will always be listed before people.
    -->
    <scm:SortDescription PropertyName="IsCompany" Direction="Descending" />
    <scm:SortDescription PropertyName="DisplayName" Direction="Ascending" />
  </CollectionViewSource.SortDescriptions>
</CollectionViewSource>
```

Die Zuordnung zwischen einem ListViewItem und einem CustomerViewModel-Objekt wird durch die ItemContainerStyle-Eigenschaft von ListView eingerichtet. Der dieser Eigenschaft zugewiesene Style wird auf jedes ListViewItem angewendet, wodurch ermöglicht wird, dass Eigenschaften in einem ListViewItem an Eigenschaften im CustomerViewModel gebunden werden. Eine wichtige Bindung in diesem Style erstellt eine Verknüpfung zwischen der IsSelected-Eigenschaft eines ListViewItem und der IsSelected-Eigenschaft eines CustomerViewModel, wie hier dargestellt:

```
<Style x:Key="CustomerItemStyle" TargetType="{x:Type ListViewItem}">
  <!--   Stretch the content of each cell so that we can
  right-align text in the Total Sales column.  -->
  <Setter Property="HorizontalContentAlignment" Value="Stretch" />
  <!--
  Bind the IsSelected property of a ListViewItem to the
  IsSelected property of a CustomerViewModel object.
  -->
  <Setter Property="IsSelected" Value="{Binding Path=IsSelected,
    Mode=TwoWay}" />
</Style>
```

Wenn ein CustomerViewModel aktiviert oder deaktiviert wird, führt dies dazu, dass sich die Summe der Gesamtumsätze aller ausgewählten Kunden ändert. Die AllCustomersViewModel-Klasse ist verantwortlich für das Verwalten dieses Werts, sodass der ContentPresenter unter ListView die richtige Zahl anzeigen kann. **Abbildung 17** zeigt, wie AllCustomersViewModel jeden Kunden dahingehend überwacht, ob er aktiviert oder deaktiviert ist, und die Ansicht benachrichtigt, dass der Anzeigewert aktualisiert werden muss.

Abbildung 17 Überwachung im Hinblick auf Aktivierungszustand

```
// In AllCustomersViewModel.cs
```

```
public double TotalSelectedSales
{
    get
    {
        return this.AllCustomers.Sum(
            custVM => custVM.IsSelected ? custVM.TotalSales : 0.0);
    }
}

void OnCustomerViewModelPropertyChanged(object sender,
    PropertyChangedEventArgs e)
{
    string IsSelected = "IsSelected";

    // Make sure that the property name we're
    // referencing is valid.  This is a debugging
    // technique, and does not execute in a Release build.
    (sender as CustomerViewModel).VerifyPropertyName(IsSelected);

    // When a customer is selected or unselected, we must let the
    // world know that the TotalSelectedSales property has changed,
    // so that it will be queried again for a new value.
    if (e.PropertyName == IsSelected)
        this.OnPropertyChanged("TotalSelectedSales");
}
```

Die wird Benutzeroberfläche an die TotalSelectedSales-Eigenschaft gebunden und wendet die Währungsformatierung an. Das ViewModel-Objekt könnte anstelle der Ansicht die Währungsformatierung anwenden, indem es einen String-Wert statt eines Double-Werts aus der TotalSelectedSales-Eigenschaft zurückgibt. Die ContentStringFormat-Eigenschaft des ContentPresenter wurde in .NET Framework 3.5 SP1 hinzugefügt. Wenn Sie es mit einer älteren WPF-Version zu tun haben, müssen Sie die Währungsformatierung im Code anwenden:

```
<!-- In AllCustomersView.xaml -->
<StackPanel Orientation="Horizontal">
  <TextBlock Text="Total selected sales: " />
  <ContentPresenter
    Content="{Binding Path=TotalSelectedSales}"
    ContentStringFormat="c"
  />
</StackPanel>
```

Zusammenfassung

WPF hat Anwendungsentwicklern viel zu bieten. Um zu erfahren, wie die Leistungsstärke genutzt werden kann, ist eine neue Denkweise erforderlich. Das Model-View-ViewModel-Muster ist ein einfacher und effektiver Satz von Richtlinien für das Entwerfen und Implementieren einer WPF-Anwendung. Es ermöglicht Ihnen, eine starke Trennung zwischen Daten, Verhalten und Darstellung zu erstellen, sodass das Chaos in der Softwareentwicklung leichter kontrolliert werden kann.

Ich möchte John Gossman für seine Hilfe bei diesem Artikel danken.

Josh Smith liegt die Verwendung von WPF zur Erzielung hervorragender Benutzerfunktionalität sehr am Herzen. Für seine Arbeit in der WPF-Community wurde ihm der Titel eines Microsoft MVP zuerkannt. Josh Smith ist für Infragistics in der Experience Design Group tätig. Wenn er nicht an einem Computer sitzt, spielt er gern Klavier, liest Bücher und Artikel über Geschichte und erkundet gemeinsam mit seiner Freundin New York. Sie finden seinen Blog unter joshsmithonwpf.wordpress.com.