

MVVM

Model – View – ViewModel

Ein Architekturmuster für die Präsentationsschicht

Handout zum Vortrag – Hintergrund, Aufbau, Abgrenzung, Praxis und Ausblick

IT-Umschulung Fachinformatiker/in Anwendungsentwicklung und Systemintegration

So arbeiten Sie mit diesem Handout

Das Handout enthält alle Inhalte des Vortrags in ausführlicher Form und eignet sich zum Nacharbeiten und zur Prüfungsvorbereitung.

Die Codebeispiele stammen aus der Demo-Anwendung „Nordfracht Sendungserfassung“ (Solution [Architekturmuster_MVVM_Demo.sln](#)). Am Ende finden Sie ein Glossar und Wiederholungsfragen.

Aussagen, die mit **(ungeprüft)** markiert sind, sollten Sie vor einer Weiterverwendung gegen eine aktuelle Quelle prüfen.

Inhalt

- 1 Warum überhaupt ein Architekturmuster?
- 2 Geschichte: von MVC zu MVVM
- 3 Aufbau von MVVM
- 4 Technische Bausteine in WPF
- 5 Regeln und bewährte Praktiken
- 6 Abgrenzung: MVC, MVP, MVVM, MVU
- 7 Verbreitung, Frameworks und Werkzeuge
- 8 Einsatzmöglichkeiten und Grenzen
- 9 Ausblick
- 10 Die Demo-Anwendung in vier Stufen
- 11 Glossar
- 12 Wiederholungsfragen
- 13 Weiterführende Quellen

1 Warum überhaupt ein Architekturmuster?

Wer zum ersten Mal eine grafische Oberfläche mit WPF oder WinForms baut, geht meist den naheliegenden Weg: Man zieht Steuerelemente auf das Fenster, klickt doppelt auf einen Button und schreibt die gesamte Logik in den erzeugten Click-Handler. Das funktioniert – für kleine Programme sogar erstaunlich gut. Mit wachsender Größe entsteht jedoch ein Code, der im Vortrag als „**Code-Behind-Monster**“ bezeichnet wird.

Typische Symptome sind:

- **Keine Testbarkeit:** Die Geschäftsregeln stecken in Ereignisbehandlern. Um sie zu prüfen, muss man das Programm starten und klicken.
- **Verklebung von UI und Logik:** Validierungen melden sich per `MessageBox`, Berechnungen lesen direkt aus Textfeldern.
- **Kein echtes Datenmodell:** Oft sind die „Daten“ nur der angezeigte Text. In der Demo wird das Gesamtgewicht aus dem Anzeigetext der Liste zurückgerechnet – ein Empfänger mit einem Senkrechtstrich im Namen bringt das Programm zum Absturz.
- **Manuelle Synchronisation:** Jede Anzeige muss an jeder Stelle von Hand aktualisiert werden. Wird ein Aufruf vergessen, zeigt die Oberfläche falsche Werte.
- **Schlechte Teamarbeit:** Gestaltung und Programmierung lassen sich nicht trennen, weil beides in derselben Datei hängt.

Die Antwort darauf ist ein altes Prinzip der Softwaretechnik: **Separation of Concerns** (Trennung der Zuständigkeiten). Darstellung, Präsentationslogik und fachliche Logik sollen in getrennten Bausteinen liegen, die möglichst wenig voneinander wissen. MVVM ist eine konkrete Umsetzung dieses Prinzips, zugeschnitten auf Oberflächen-Technologien mit leistungsfähigem **DataBinding**.

Merksatz

Ein Architekturmuster macht ein Programm nicht schneller oder schöner. Es macht es **änderbar und testbar** – und das entscheidet über die Kosten, sobald ein Programm länger lebt als ein paar Wochen.

Architekturmuster oder Entwurfsmuster?

MVVM ist ein **Architekturmuster** (englisch „architectural pattern“), genauer ein UI- bzw. Präsentationsmuster: Es legt die Grundstruktur der **gesamten Präsentationsschicht** fest. Die 23 **GoF-Entwurfsmuster** (englisch „design patterns“) wie Observer oder Strategy regeln dagegen das Zusammenspiel **weniger Klassen**.

Beide Ebenen hängen zusammen: MVVM ist aus Entwurfsmustern gebaut – **Observer** (`INotifyPropertyChanged`), **Command** (`ICommand`) und für die Kommunikation zwischen ViewModels oft **Mediator** (Messenger).

Im Alltag – und auch in mancher Dokumentation – wird „design pattern“ locker als Oberbegriff für beides verwendet. Fachlich sauber ist die Unterscheidung nach der Reichweite.

2 Geschichte: von MVC zu MVVM

MVVM ist nicht aus dem Nichts entstanden. Es ist das vorläufige Ende einer Entwicklung, die vor über 45 Jahren begann. Wer die Vorgänger kennt, versteht, warum MVVM so aufgebaut ist.

2.1 MVC – Model View Controller (1978/79)

Das Muster **Model-View-Controller** wurde von **Trygve Reenskaug** in den Jahren 1978/79 am Forschungszentrum **Xerox PARC** entwickelt und in **Smalltalk-80** umgesetzt. Die Grundidee: Das **Model** hält die Daten, die **View** stellt sie dar, und der **Controller** verarbeitet Eingaben von Maus und Tastatur. Ändert sich das Model, benachrichtigt es die View (Beobachter-Muster), die sich daraufhin neu zeichnet.

Wichtig für das Verständnis: Das ursprüngliche MVC war sehr feingranular gedacht – jedes einzelne Steuerelement hatte sein eigenes Trio aus Model, View und Controller. Das heute bekannte **Web-MVC** (etwa ASP.NET MVC oder Ruby on Rails) ist eine spätere Neuinterpretation: Der Controller nimmt eine HTTP-Anfrage entgegen, holt Daten aus dem Model und übergibt sie an eine View, die HTML erzeugt. Der Begriff MVC wird deshalb in der Literatur recht unterschiedlich verwendet.

2.2 MVP – Model View Presenter (1996)

Mitte der 1990er-Jahre entstand das Muster **Model-View-Presenter** in der C++- und Smalltalk-Welt. Der Controller wird hier zum **Presenter**, der die gesamte Präsentationslogik übernimmt. Die View wird möglichst „dumm“: Sie leitet Eingaben an den Presenter weiter und lässt sich von ihm befüllen.

Entscheidend ist: Der Presenter **kennt die View** – allerdings nur über ein Interface (z. B. **IKundenView**). Dadurch lässt sich der Presenter mit einer Attrappe der View testen. Später unterschied man zwei Varianten: Bei der **Passive View** setzt der Presenter jedes einzelne Feld der View selbst; beim **Supervising Controller** übernimmt einfaches DataBinding einen Teil der Arbeit. MVP war lange das bevorzugte Muster für Windows Forms und für frühe Android-Anwendungen.

2.3 Presentation Model (2004)

2004 beschrieb **Martin Fowler** das Muster **Presentation Model**. Die Idee: Man baut eine Klasse, die den **Zustand und das Verhalten der Oberfläche** abbildet, aber völlig unabhängig von konkreten UI-Klassen ist. Ob ein Button aktiv ist oder welcher Eintrag ausgewählt ist, steht als ganz normale Eigenschaft in dieser Klasse. Die View synchronisiert sich mit dem Presentation Model. Im Gegensatz zu MVP kennt das Presentation Model die View **nicht**.

2.4 MVVM – Model View ViewModel (2005)

John Gossman, damals Architekt bei **Microsoft** für die neue Oberflächentechnologie **WPF** (Codename „Avalon“) und später Silverlight, stellte MVVM im **Oktober 2005** in einem Blogbeitrag vor. MVVM ist im Kern das Presentation Model, angepasst an die Möglichkeiten von WPF. Das **ViewModel** entspricht dem Presentation Model; die lästige Synchronisation mit der View übernimmt das mächtige **DataBinding** von WPF automatisch.

Ein wesentlicher Beweggrund war die Zusammenarbeit von **Designern und Entwicklern**: Mit Expression Blend sollten Designer die Oberfläche in XAML gestalten können, ohne Programmcode anzufassen, während Entwickler unabhängig davon die Logik im ViewModel schreiben.

2.5 Überblick als Zeitstrahl

Jahr	Ereignis
1978/79	MVC für Smalltalk-80 (Trygve Reenskaug)
1996	MVP – der Presenter steuert eine passive View
2004	Presentation Model (Martin Fowler)
2005	MVVM für WPF (John Gossman, Microsoft)
2006	Aufteilung von MVP in „Passive View“ und „Supervising Controller“
2010	Knockout.js bringt MVVM in den Browser
2012	Die Sprache Elm führt MVU ein
2017	Android erhält eine eigene ViewModel -Klasse
2022	CommunityToolkit.Mvvm 8.0 mit Source Generators

3 Aufbau von MVVM

MVVM besteht aus drei Schichten. Entscheidend ist nicht nur, **was** jede Schicht tut, sondern vor allem, **was sie über die anderen weiß**.

Schicht	Aufgabe	Kennt	Beispiel aus der Demo
Model	Fachliche Daten, Geschäftsregeln, Datenzugriff (Datenbank, Dateien, Webdienste)	nichts von der Oberfläche	Klasse <code>Sendung</code> mit Regel <code>IstGewichtGueltig()</code>
ViewModel	Zustand und Verhalten der Oberfläche: Eigenschaften zum Anzeigen, Befehle, Validierung, abgeleitete Werte	das Model – aber nicht die View	<code>MainViewModel</code> mit <code>Empfaenger</code> , <code>HinzufuegenCommand</code> , <code>StatusText</code>
View	Darstellung in XAML, Layout, Stile, Animationen	das ViewModel nur über <code>DataContext</code> und Bindings	<code>MainWindow.xaml</code>

3.1 Die Abhängigkeitsrichtung

Die Abhängigkeiten zeigen immer nur in eine Richtung: **View → ViewModel → Model**. Das ViewModel ruft die View niemals direkt auf, es kennt nicht einmal ihren Typ. Genau das macht es austauschbar und testbar: Ein Unit-Test kann die Rolle der View übernehmen, indem er Eigenschaften setzt und Befehle ausführt.

3.2 Wie die Schichten zusammenspielen

Ein typischer Ablauf in der Demo-Anwendung sieht so aus:

1. Der Benutzer tippt einen Empfänger in das Textfeld.
2. Das **Binding** schreibt den Text in die Eigenschaft `Empfaenger` des ViewModels.
3. Das ViewModel prüft, ob jetzt hinzugefügt werden darf, und meldet dem Befehl `HinzufuegenCommand` eine mögliche Änderung (`CanExecuteChanged`).
4. WPF fragt `CanExecute()` erneut ab und aktiviert den Button – ohne eine Zeile Code in der View.
5. Nach dem Klick fügt der Befehl eine neue `Sendung` in die `ObservableCollection` ein. Die Liste meldet die Änderung, die `ListBox` zeigt den neuen Eintrag, und das ViewModel meldet per `PropertyChanged`, dass sich `StatusText` geändert hat.

Der Datenfluss ist also **bidirektional**: Eingaben fließen über Bindings in das ViewModel, Änderungen im ViewModel fließen über Benachrichtigungen zurück in die View.

Denkmodell

Stellen Sie sich das ViewModel als „**Oberfläche ohne Pixel**“ vor. Alles, was man auf dem Bildschirm sieht oder tun kann, existiert im ViewModel als Eigenschaft oder Befehl – nur eben ohne Farben, Schriften und Layout.

4 Technische Bausteine in WPF

4.1 DataContext und Binding

Jedes WPF-Element besitzt die Eigenschaft `DataContext`. Sie wird an alle Kind-Elemente vererbt. Setzt man den DataContext des Fensters auf das ViewModel, können sich alle Steuerelemente darin per `{Binding Eigenschaft}` auf dessen Eigenschaften beziehen.

```
<TextBox Text="{Binding Empfaenger, UpdateSourceTrigger=PropertyChanged}" />
<Button Content="Hinzufügen" Command="{Binding HinzufuegenCommand}" />
<ListBox ItemsSource="{Binding Sendungen}"
         SelectedItem="{Binding AusgewaehlteSendung, Mode=TwoWay}" />
```

Modus	Richtung	Typischer Einsatz
OneWay	ViewModel → View	Anzeigen, z. B. <code>TextBlock</code>
TwoWay	ViewModel ↔ View	Eingaben, z. B. <code>TextBox</code> , <code>SelectedItem</code>
OneTime	einmalig beim Laden	Werte, die sich nie ändern
OneWayToSource	View → ViewModel	selten, z. B. Größe eines Elements auslesen

Mit `UpdateSourceTrigger=PropertyChanged` wird bei jedem Tastendruck in das ViewModel geschrieben; der Standard bei `TextBox` ist `LostFocus`.

4.2 INotifyPropertyChanged

Damit die View merkt, dass sich eine Eigenschaft im ViewModel geändert hat, implementiert das ViewModel die Schnittstelle `INotifyPropertyChanged` und löst das Ereignis `PropertyChanged` aus. Von Hand geschrieben (Stufe 2 der Demo) sieht das so aus:

```
public string Empfaenger
{
    get { return _empfaenger; }
    set
    {
        if (_empfaenger == value) return;
        _empfaenger = value;
        OnPropertyChanged("Empfaenger"); // String: Tippfehler unbemerkt!
        HinzufuegenCommand.RaiseCanExecuteChanged();
    }
}
```

Mit einer Basisklasse und dem Attribut `[CallerMemberName]` (Stufe 3) schrumpft das auf wenige Zeilen, und der Property-Name wird vom Compiler eingesetzt:

```
protected bool SetProperty<T>(ref T feld, T wert, [CallerMemberName] string name = null)
{
    if (EqualityComparer<T>.Default.Equals(feld, wert)) return false;
    feld = wert;
    OnPropertyChanged(name);
    return true;
}

public string Empfaenger
{
    get { return _empfaenger; }
    set { if (SetProperty(ref _empfaenger, value))
        HinzufuegenCommand.RaiseCanExecuteChanged(); }
}
```

4.3 ICommand

Statt Click-Handlern verwendet MVVM **Befehle**. Ein Befehl implementiert **ICommand** mit den Methoden **Execute()** (was passiert beim Klick) und **CanExecute()** (darf überhaupt geklickt werden). WPF deaktiviert einen Button automatisch, solange **CanExecute()** **false** liefert. Da WPF keine fertige Implementierung für eigene Methoden mitbringt, schreibt man eine kleine Hilfsklasse, meist **DelegateCommand** oder **RelayCommand** genannt:

```
public class DelegateCommand : ICommand
{
    private readonly Action _execute;
    private readonly Func<bool> _canExecute;

    public event EventHandler CanExecuteChanged;
    public bool CanExecute(object p) { return _canExecute == null || _canExecute(); }
    public void Execute(object p) { if (CanExecute(p)) _execute(); }
    public void RaiseCanExecuteChanged()
    {
        var h = CanExecuteChanged; if (h != null) h(this, EventArgs.Empty);
    }
}
```

4.4 ObservableCollection<T>

Für Listen verwendet man **ObservableCollection<T>**. Sie implementiert **INotifyCollectionChanged** und meldet jedes Hinzufügen und Entfernen an gebundene Steuerelemente. Eine normale **List<T>** würde zwar anzeigen, Änderungen danach aber nicht. Achtung: Die Collection meldet nur Änderungen **an der Liste**, nicht an Eigenschaften der enthaltenen Objekte – dafür müssen die Elemente selbst **INotifyPropertyChanged** implementieren.

4.5 DataTemplate, Converter und Validierung

- **DataTemplate:** legt fest, wie ein Objekt dargestellt wird, z. B. jede **Sendung** in der Liste mit fettem Empfänger und formatiertem Gewicht.
- **IValueConverter:** wandelt Werte beim Binding um, etwa **bool** in **Visibility**. So bleiben UI-Typen wie **Visibility** aus dem ViewModel heraus.
- **Validierung:** Der WPF-Standard ist **INotifyDataErrorInfo** (älter: **IDataErrorInfo**). Das ViewModel meldet Fehler pro Eigenschaft, die View zeigt sie z. B. mit rotem Rahmen an. Die Demo nutzt zur Vereinfachung eine eigene Eigenschaft **Fehlermeldung**.

5 Regeln und bewährte Praktiken

Die folgenden Regeln sind keine Formalie. Jede einzelne schützt eine bestimmte Eigenschaft des Musters – meist die Testbarkeit.

1. **Keine UI-Typen im ViewModel.** [Window](#), [MessageBox](#), [Brush](#), [Visibility](#) oder Steuerelemente haben im ViewModel nichts zu suchen. Sobald das ViewModel ein Fenster öffnet, lässt es sich nicht mehr automatisiert testen.
2. **Der Lackmustest ist der Unit-Test.** Ein ViewModel ist dann gut, wenn man es ohne Fenster instanziiert, Eigenschaften setzt, Befehle ausführt und das Ergebnis prüfen kann. Die Demo enthält dafür sieben MSTest-Tests.
3. **Code-Behind ist nicht verboten.** Reine Darstellungslogik – Fokus setzen, Animationen, Scrollen – darf im Code-Behind stehen. Die Regel lautet: keine **Geschäfts- oder Präsentationslogik** im Code-Behind. „Null Zeilen Code-Behind um jeden Preis“ führt oft zu unnötig komplizierten Konstruktionen.
4. **Dialoge und Navigation über Dienste.** Braucht das ViewModel eine Bestätigung, ruft es eine Schnittstelle wie [IDialogService](#) auf. Die echte Anwendung liefert eine Umsetzung mit [MessageBox](#), der Test eine Attrappe (Fake), die eine vorgegebene Antwort liefert.
5. **Abhängigkeiten von außen übergeben.** Dienste gelangen über den Konstruktor ins ViewModel (**Dependency Injection**). Verdrahtet wird an genau einer Stelle, dem **Composition Root** – in der Demo [App.xaml.cs](#), in größeren Projekten ein DI-Container wie [Microsoft.Extensions.DependencyInjection](#).
6. **Geschäftsregeln ins Model.** Die Regel „maximal 31,5 kg“ steht einmal in der Klasse [Sendung](#), nicht verteilt über Click-Handler und ViewModels.
7. **Kommunikation zwischen ViewModels lose koppeln.** Statt dass sich ViewModels gegenseitig kennen, nutzt man Dienste oder einen **Messenger** (z. B. [WeakReferenceMessenger](#) im CommunityToolkit, [EventAggregator](#) in Prism).

Typische Fehler aus der Praxis

- Property-Name im String falsch geschrieben – das Binding schweigt, nur das Ausgabefenster meldet „System.Windows.Data Error: 40“.
- [List<T>](#) statt [ObservableCollection<T>](#) – neue Einträge erscheinen nicht.
- [RaiseCanExecuteChanged\(\)](#) vergessen – der Button bleibt grau, obwohl die Eingabe gültig ist.
- Abgeleitete Eigenschaften (z. B. [StatusText](#)) werden nicht mitgemeldet – die Anzeige bleibt stehen.
- DataContext nicht gesetzt – alle Bindings laufen ins Leere, ohne Fehlermeldung.

6 Abgrenzung: MVC, MVP, MVVM, MVU

Alle vier Muster verfolgen dasselbe Ziel: die Trennung von Darstellung und Logik. Sie unterscheiden sich darin, **wer die View aktualisiert, wer wen kennt und wie die Daten fließen**.

Merkmal	MVC	MVP	MVVM	MVU
Vermittler	Controller	Presenter	ViewModel	Update-Funktion
Kennt der Vermittler die View?	teilweise	ja, über Interface	nein	nein
Wer aktualisiert die View?	View liest aus dem Model	Presenter ruft Methoden der View auf	DataBinding automatisch	View wird aus dem Zustand neu berechnet
Datenfluss	gemischt	bidirektional, manuell	bidirektional, per Binding	unidirektional
Zustand	verteilt	im Presenter	im ViewModel, veränderlich	ein zentraler, unveränderlicher Zustand
Typische Technik	ASP.NET MVC, Ruby on Rails, Spring MVC	Windows Forms, frühes Android	WPF, .NET MAUI, WinUI, Avalonia	Elm, Redux, Fabulous (F#)

6.1 MVVM gegenüber MVP

Die beiden Muster liegen am nächsten beieinander. Der Unterschied lässt sich an einer einzigen Zeile zeigen. Im MVP ruft der Presenter die View aktiv auf: `view.SetEmpfaenger("Hansen")`. Im MVVM setzt das ViewModel nur seine eigene Eigenschaft: `Empfaenger = "Hansen"` – den Rest erledigt das Binding. Das ViewModel braucht deshalb nicht einmal ein View-Interface. Der Preis: MVVM setzt ein leistungsfähiges Binding-System voraus, das Windows Forms in dieser Form nicht bietet.

6.2 MVVM gegenüber MVC

Beim klassischen Web-MVC ist der Controller der Einstiegspunkt: Er empfängt eine Anfrage, entscheidet, was zu tun ist, und wählt eine View aus. Bei MVVM ist die View der Einstiegspunkt; sie bindet sich an ein langlebiges ViewModel, das über die gesamte Lebensdauer des Fensters Zustand hält. MVC passt deshalb gut zum zustandslosen Anfrage-Antwort-Modell des Webs, MVVM zu zustandsreichen Desktop- und Mobil-Anwendungen.

6.3 MVVM gegenüber MVU

Model-View-Update stammt aus der Programmiersprache **Elm** (2012) und ist dort als „The Elm Architecture“ bekannt. Es gibt genau **einen Zustand** (Model), der **unveränderlich** ist. Benutzeraktionen erzeugen **Nachrichten**; eine reine **Update-Funktion** berechnet daraus einen neuen Zustand; eine **View-Funktion** erzeugt aus dem Zustand die komplette Oberfläche neu. Der Datenfluss läuft nur in eine Richtung: Nachricht → Update → neuer Zustand → View.

Vorteil: Das Verhalten ist sehr gut nachvollziehbar, weil es keinen verteilten, veränderlichen Zustand gibt. Nachteil: Das Denkmodell ist für Entwicklerinnen und Entwickler aus der objektorientierten Welt ungewohnt, und bei großen Oberflächen muss das Framework das Neuzeichnen geschickt optimieren. React mit Redux, SwiftUI und Jetpack Compose folgen ähnlichen Ideen.

7 Verbreitung, Frameworks und Werkzeuge

MVVM ist heute das Standardmuster überall dort, wo Oberflächen in XAML beschrieben werden, und hat darüber hinaus Spuren im Web und in der Mobilentwicklung hinterlassen.

Bereich	Technologien
Microsoft / .NET	WPF, Silverlight (eingestellt), UWP, Xamarin.Forms (Nachfolger: .NET MAUI), WinUI 3
Plattformübergreifend .NET	Avalonia UI, Uno Platform, .NET MAUI
Web	Knockout.js (2010); Vue.js nennt MVVM ausdrücklich als Inspiration; Angular folgt ähnlichen Ideen
Mobil	Android Jetpack ViewModel (seit 2017), SwiftUI mit ObservableObject bzw. @Observable

7.1 MVVM-Bibliotheken für .NET

Bibliothek	Charakter
CommunityToolkit.Mvvm	Heute De-facto-Standard, von Microsoft gepflegt. Klein, schnell, mit Source Generators ([ObservableProperty] , [RelayCommand]). Läuft über .NET Standard 2.0 auch unter .NET Framework 4.8; die Source Generators benötigen dafür aber eine neuere C#-Sprachversion als 7.3 (ungeprüft – in der Toolkit-Dokumentation nachlesen).
Prism	Großes Framework für modulare Anwendungen: Regionen, Navigation, Module, EventAggregator. Neuere Versionen stehen unter einer kommerziellen Lizenz (ungeprüft – Lizenzbedingungen vor Einsatz prüfen).
Caliburn.Micro	Setzt stark auf Namenskonventionen („Convention over Configuration“), z. B. bindet ein Button namens Speichern automatisch an die Methode Speichern() .
ReactiveUI	Kombiniert MVVM mit Reactive Extensions (Rx). Mächtig, aber mit steiler Lernkurve.
MVVM Light	Früher sehr verbreitet, wird nicht mehr weiterentwickelt. Empfohlener Nachfolger ist das CommunityToolkit.
Eigene Basisklassen	Zwei Klassen (ViewModelBase, DelegateCommand) genügen für viele Projekte. Didaktisch wertvoll, weil nichts „magisch“ passiert.

7.2 Werkzeuge

- **XAML-Hot-Reload** in Visual Studio: Änderungen an der Oberfläche während der Laufzeit.
- **Ausgabefenster / XAML-Bindungsfehler-Fenster**: zeigt fehlerhafte Bindings an – bei der Fehlersuche die erste Anlaufstelle.
- **Designzeit-DataContext** ([d:DataContext](#)): liefert Beispieldaten für den Designer, ohne das Programm zu starten.

8 Einsatzmöglichkeiten und Grenzen

Vorteile	Nachteile
Testbarkeit: Präsentationslogik ist ohne Oberfläche per Unit-Test prüfbar.	Boilerplate: Ohne Hilfsmittel viel gleichförmiger Code für Properties und Commands.
Trennung: Design und Logik können parallel und unabhängig bearbeitet werden.	Laufzeitfehler: Falsche Binding-Pfade fallen nicht beim Kompilieren auf.
Wiederverwendung: Ein ViewModel kann mehrere Views bedienen, z. B. Desktop und Tablet.	Lernkurve: Binding, Commands, DI und Services müssen erst verstanden werden.
Wartbarkeit: Änderungen an der Oberfläche berühren die Logik nicht und umgekehrt.	Overhead: Bei sehr kleinen Programmen steht der Aufwand in keinem Verhältnis zum Nutzen.
Werkzeugunterstützung: Designer, Hot Reload und Toolkit sind auf MVVM ausgelegt.	Speicher und Leistung: Viele Bindings und vergessene Ereignisabonnements können bei großen Anwendungen Probleme verursachen.

8.1 Wann MVVM sinnvoll ist

- Desktop- und Mobil-Anwendungen mit vielen Zuständen (Formulare, Masken, Dashboards)
- Software, die über Jahre gepflegt und erweitert wird
- Projekte mit Anforderungen an automatisierte Tests
- Teams, in denen Gestaltung und Programmierung getrennt laufen

8.2 Wann weniger mehr ist

- Kleine Hilfsprogramme mit einem Fenster und wenigen Feldern
- Wegwerf-Prototypen zur Klärung einer Idee
- Technologien ohne leistungsfähiges DataBinding (z. B. klassische Windows Forms – dort passt MVP besser)

Einschätzung

MVVM ist ein Werkzeug, kein Glaubensbekenntnis. Ein häufiger Anfängerfehler ist das Überarchitekturen: DI-Container, Messenger und fünf Schichten für ein Programm mit zwei Textfeldern. Beginnen Sie mit ViewModelBase und DelegateCommand – und führen Sie weitere Bausteine erst ein, wenn ein konkretes Problem sie verlangt.

9 Ausblick

9.1 Source Generators beseitigen den Boilerplate

Seit Version 8 (2022) erzeugt das **CommunityToolkit.Mvvm** den wiederkehrenden Code beim Kompilieren. Aus einem Feld mit **[ObservableProperty]** wird eine vollständige Eigenschaft mit Änderungsbenachrichtigung, aus einer Methode mit **[RelayCommand]** ein fertiger Befehl. Die Klasse muss dafür **partial** sein. Stufe 4 der Demo zeigt den Vergleich:

```
public partial class MainViewModel : ObservableObject
{
    [ObservableProperty]
    [NotifyCanExecuteChangedFor(nameof(HinzufuegenCommand))]
    private string _empfaenger = ""; // -> Property Empfaenger

    [RelayCommand(CanExecute = nameof(KannHinzufuegen))]
    private void Hinzufuegen() { ... } // -> HinzufuegenCommand
}
```

In neueren Toolkit-Versionen mit aktuellem C# können statt Feldern auch **partielle Eigenschaften** annotiert werden (**ungeprüft – Versions- und Sprachanforderungen in der Toolkit-Dokumentation nachlesen**).

9.2 Compiled Bindings

Klassische WPF-Bindings werden zur Laufzeit per Reflection aufgelöst. **Compiled Bindings** – **x:Bind** in UWP/WinUI, **x:DataType** in .NET MAUI und Avalonia – werden beim Kompilieren geprüft. Ein falscher Property-Name wird damit zum Compilerfehler, und die Leistung steigt. Für klassisches WPF gibt es dieses Merkmal nicht.

9.3 Unidirektionale und deklarative Oberflächen

In der gesamten Branche ist ein Trend zu **deklarativen Oberflächen mit unidirektionalem Datenfluss** erkennbar: React, SwiftUI und Jetpack Compose beschreiben die Oberfläche als Funktion des Zustands. Im .NET-Umfeld verfolgt **Fabulous (F#)** diesen MVU-Ansatz; Microsoft hat mit dem Projekt „Comet“ MVU für .NET erprobt (**ungeprüft – Projektstatus unklar**). Blazor nutzt ein Komponentenmodell, das ebenfalls eher diesem Denken folgt.

9.4 Fazit

MVVM wird in absehbarer Zeit nicht verschwinden. Solange Oberflächen in XAML beschrieben werden – WPF, WinUI, MAUI, Avalonia, Uno – bleibt es das Standardmuster. Es bekommt aber Konkurrenz durch unidirektionale Ansätze, und die Werkzeuge nehmen einen immer größeren Teil der Handarbeit ab. Wer MVVM verstanden hat, versteht die Kernidee aller modernen UI-Architekturen: **Die Oberfläche ist eine Abbildung eines Zustands, und dieser Zustand gehört in testbaren Code.**

10 Die Demo-Anwendung in vier Stufen

Die Solution `Architekturmuster_MVVM_Demo.sln` enthält dieselbe kleine Anwendung der Nordfracht Logistik GmbH in vier Ausbaustufen: Sendungen mit Empfänger und Gewicht erfassen, in einer Liste anzeigen, löschen und das Gesamtgewicht berechnen.

Stufe	Projekt	Kernaussage
1	<code>Stufe1_CodeBehindMonster</code>	Alles im Code-Behind. Funktioniert – bis ein Empfänger einen Senkrechtstrich enthält. Nicht testbar.
2	<code>Stufe2_MvvmHandarbeit</code>	Saubere Trennung, Code-Behind leer. <code>INotifyPropertyChanged</code> in jeder Property von Hand, Namen als Strings.
3	<code>Stufe3_MvvmBasisklasse</code>	<code>ViewModelBase</code> mit <code>SetProperty</code> , <code>IDialogService</code> für die Löschabfrage, Composition Root, sieben Unit-Tests .
4	<code>Stufe4_MvvmToolkit</code>	CommunityToolkit.Mvvm auf .NET 8. Gleiche Funktion, ein Bruchteil des Codes. Das XAML entspricht Stufe 3 (nur Klassenname und Fenstertitel geändert).

10.1 Aufgaben zum Selbststudium

1. Ergänzen Sie in Stufe 3 eine Eigenschaft `Prioritaet` (Standard/Express) mit einer `ComboBox`. Express-Sendungen dürfen höchstens 20 kg wiegen. Wo gehört diese Regel hin?
2. Schreiben Sie einen Unit-Test, der prüft, dass `StatusText` nach dem Hinzufügen zweier Sendungen „2 Sendungen, gesamt ...“ lautet.
3. Ersetzen Sie die Eigenschaft `Fehlermeldung` durch eine Umsetzung von `INotifyDataErrorInfo`, sodass das Textfeld einen roten Rahmen erhält.
4. Überlegen Sie: Warum wäre ein `record` für die Klasse `Sendung` in Stufe 4 problematisch? (Hinweis: Wertgleichheit und `Remove()`.)

11 Glossar

Begriff	Erklärung
Binding	Automatische Verbindung zwischen einer Eigenschaft der View und einer Eigenschaft des ViewModels.
Code-Behind	Die C#-Datei, die zu einer XAML-Datei gehört (<code>MainWindow.xaml.cs</code>).
Command / ICommand	Objekt, das eine ausführbare Aktion samt Freigabeprüfung (<code>CanExecute</code>) kapselt.
Composition Root	Die eine Stelle im Programm, an der alle konkreten Klassen erzeugt und verdrahtet werden.
DataContext	Objekt, auf das sich die Bindings eines Elements und seiner Kinder beziehen.
DataTemplate	Vorlage, die festlegt, wie ein Datenobjekt dargestellt wird.
Dependency Injection	Abhängigkeiten werden von außen übergeben statt intern erzeugt.
Fake / Attrappe	Einfache Test-Implementierung einer Schnittstelle mit vorgegebenem Verhalten.
INotifyPropertyChanged	Schnittstelle, über die ein Objekt Änderungen seiner Eigenschaften meldet.
Messenger	Vermittler für Nachrichten zwischen ViewModels, die sich nicht kennen sollen.
ObservableCollection	Liste, die Hinzufügen und Entfernen von Elementen meldet.
Presentation Model	Von Martin Fowler beschriebenes Muster, Vorläufer des ViewModels.
Separation of Concerns	Prinzip, unterschiedliche Zuständigkeiten in getrennte Bausteine zu legen.
Source Generator	Compiler-Erweiterung, die beim Übersetzen zusätzlichen Quellcode erzeugt.
Unidirektionaler Datenfluss	Daten fließen nur in eine Richtung: Aktion → neuer Zustand → Oberfläche.
ValueConverter	Klasse, die Werte beim Binding umrechnet, z. B. <code>bool</code> in <code>Visibility</code> .

12 Wiederholungsfragen

Die Antworten finden Sie in den angegebenen Kapiteln.

1. Welches Problem soll MVVM lösen? Nennen Sie drei Symptome einer Anwendung ohne Architekturmuster. (Kap. 1)
2. Wer hat MVVM wann und für welche Technologie vorgestellt? Aus welchem älteren Muster ist es hervorgegangen? (Kap. 2)
3. Beschreiben Sie die Aufgaben von Model, View und ViewModel. Welche Schicht kennt welche? (Kap. 3)
4. Warum darf das ViewModel keine `MessageBox` anzeigen? Wie löst man eine Sicherheitsabfrage stattdessen? (Kap. 5)
5. Wozu dient `INotifyPropertyChanged`? Was passiert, wenn das Ereignis nicht ausgelöst wird? (Kap. 4)
6. Erklären Sie `Execute()` und `CanExecute()` eines Commands anhand eines Beispiels. (Kap. 4)
7. Warum reicht eine `List<T>` für eine gebundene Liste nicht aus? (Kap. 4)
8. Worin unterscheiden sich MVP und MVVM? Formulieren Sie den Unterschied an einem Codebeispiel. (Kap. 6)
9. Beschreiben Sie den Datenfluss bei MVU. Was bedeutet „unidirektional“? (Kap. 6)
10. Nennen Sie je zwei Vor- und Nachteile von MVVM und eine Situation, in der Sie darauf verzichten würden. (Kap. 8)
11. Was leisten Source Generators im `CommunityToolkit.Mvvm`? (Kap. 9)
12. Was ist mit dem „Lackmustest“ für ein ViewModel gemeint? (Kap. 5)

13 Weiterführende Quellen

- John Gossman: „Introduction to Model/View/ViewModel pattern for building WPF apps“, Blogbeitrag, Oktober 2005.
- Martin Fowler: „GUI Architectures“, 2006, martinfowler.com – der beste Überblick über MVC, MVP und Verwandte.
- Microsoft Learn: Dokumentation zu „CommunityToolkit.Mvvm“ und „Data binding overview (WPF .NET)“.