VERTICAL SLICE ARCHITECTURE

Ein Stück vom Kuchen

Mit Feature Slices die Phasen von Lernen und Fehlersuche im Alltag verkürzen.

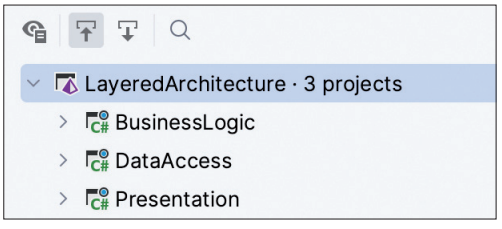
Wer kennt es nicht: Man kommt in ein neues Team oder erbt ein Projekt und sieht sich mit der Aufgabe konfrontiert, sich in die Code-Basis einzuarbeiten. Trotz Dokumentation der Fachlichkeit, Clean Code und Clean oder Hexagonal Architecture dauert es eine Weile, bis man sich im Code zurechtfindet und heimisch fühlt.

Woran das unter Umständen liegt und wie man dieses und andere Probleme umgehen kann, zeigt der vorliegende Artikel anhand eines Architektur-Ansatzes, der manchem Leser gar nicht so unbekannt vorkommen könnte.

Mein Code, der hat sechs Ecken

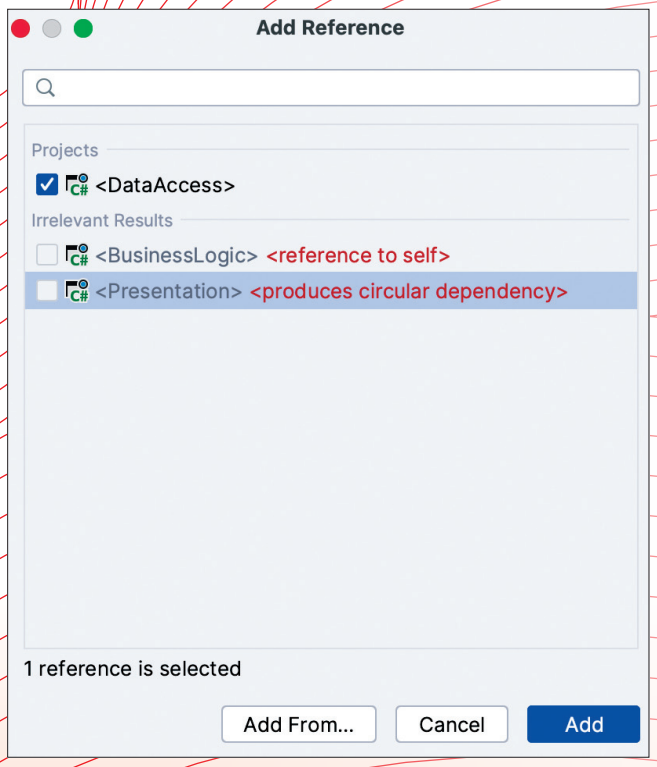
Fast alle populären Architekturstile der letzten 15 Jahre verfolgen das gleiche Ziel: klare Trennung von Zuständigkeiten und Vermeidung von Abhängigkeiten, die sich sprichwörtlich kreuz und quer durch den Code ziehen. Vor einigen Jahren wurde deshalb der Monolith als Übeltäter ausgemacht und man pries Microservices als Erlösung an. Was am Ende häufig blieb, war Ernüchterung.

3-Schichten-Architektur mit übergreifenden Aspekten und Datenbank (Bild 1)



3-Schichten-Architektur als .NET-Solution (Bild 2)

Foto: Shutterstock / Svjatoslav Andreichyn



Vermeidung zirkulärer Abhängigkeit für das BusinessLogic-Projekt (Bild 3)

Bevor wir einen alternativen Ansatz betrachten, wollen wir zunächst einige der bisherigen Architekturstile rekapitulieren. Ab Beginn der 2000er-Jahre (obschon weit vor dem Jahr 2000 entstanden) war die Schichten-Architektur das Mittel der Wahl, wenn es darum ging, Code nach Zuständigkeiten zu organisieren und Abhängigkeiten zu bändigen.

Üblich sind die drei Schichten Präsentation, Logik und Datenhaltung. Darüber hinaus gibt es schichtenübergreifende Aspekte wie Logging, Monitoring und Sicherheit. Bild 1 zeigt dies exemplarisch inklusive der Datenbank.

Das Bild enthält ebenfalls Richtungspfeile, die anzeigen, wie Code aufgerufen werden darf – ausgehend von der jeweils aktuellen Schicht nur Code aus der darunterliegenden. Daten fließen dann zurück in die darüberliegende Schicht, bis sie beim Anwender ankommen.

Während die Schichten die Trennung von Zuständigkeit im Code adressieren, versucht man Abhängigkeiten zwischen Code-Einheiten durch die Regel, in welcher Richtung Code aufgerufen werden darf, in den Griff zu bekommen. Technische Unterstützung durch den Compiler erhält man, indem die jeweilige Schicht einem Code-Modul entspricht. In .NET ist dies ein Projekt einer Solution (Bild 2). Da Projektabhängigkeiten nur in eine Richtung zeigen können, vermeidet man zirkuläre Abhängigkeiten, was Entwicklern die Einhaltung der Struktur auferlegt. Dies ist in Bild 3 zu sehen.

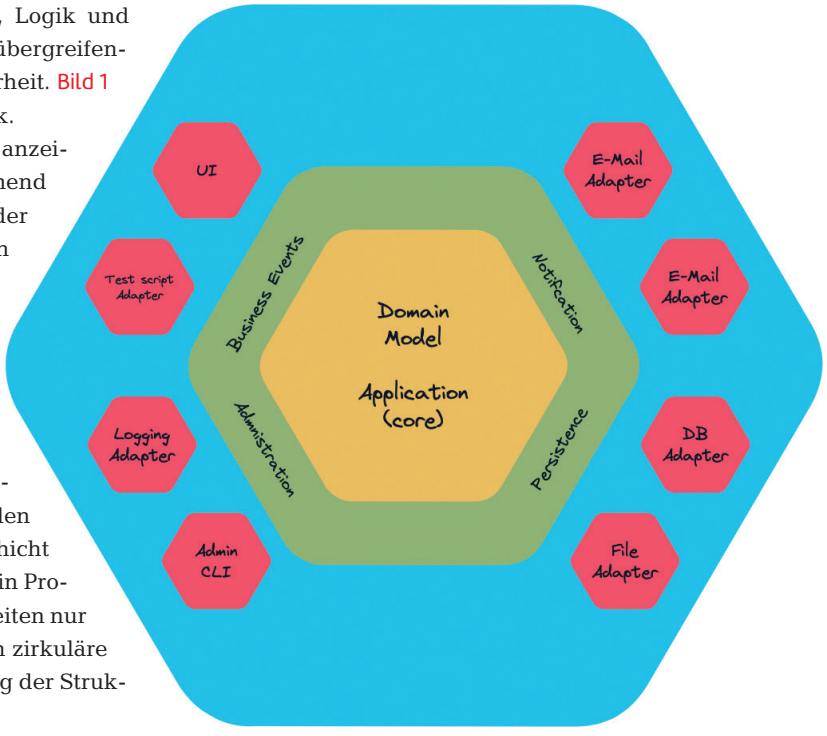
Anfang 2005 wurde ein weiterer Architekturstil von Alistair Cockburn vorgestellt: die hexagonale Architektur. Ein weiterer Name, der sich hierfür eingebürgert hat, lautet Ports-and-Adapters-Architektur. Die beiden Namen in Kombination sind bereits recht aufschlussreich im Hinblick darauf, wie dieses Muster funktioniert:

Eine hexagonale Architektur besteht aus drei Schichten, die in konzentrischen Sechsecken angeordnet sind. Der wichtigste Teil ist das Domänenmodell, das die gesamte Logik und die Regeln der Anwendung enthält. In der Domäne werden keine technologischen Belange wie HTTP-Kontexte oder Datenbankaufrufe referenziert. Damit können technologische Änderungen ohne Auswirkungen auf die Domäne vorgenommen werden.

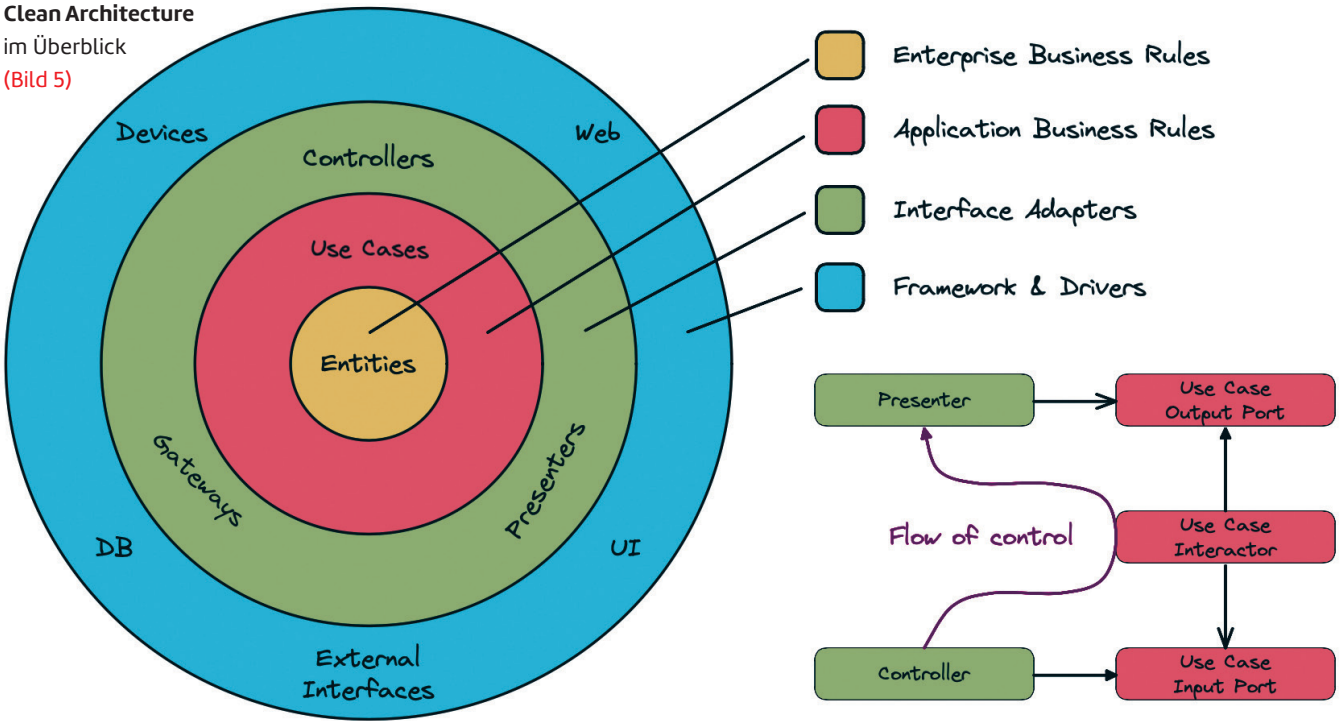
Um das Domänenmodell herum befindet sich die Ports-Schicht. Sie empfängt alle Anfragen, die einem Anwendungsfall entsprechen, der die Arbeit im Domänenmodell orchestriert. Die Ports-Schicht bildet eine Grenze zwischen der Domäne auf der Innenseite und externen Komponenten auf der Außenseite.

Außerhalb der Ports befindet sich die Adapter-Schicht. Dies ist der Technologie-Stack, der Eingaben in einem definierten Format entgegennimmt und Ausgaben erzeugt. Dies kann zum Beispiel ein HTTP-Request sein. Adapter enthalten keine Domänenlogik. Die Aufgabe eines Adapters ist eine technische Transformation zwischen der Außenwelt und der internen Domäne. Jeder Adapter, der sich an das Protokoll eines Ports implementiert, kann diesen verwenden. Mehrere Adapter können denselben Port verwenden. So lassen sich zum Beispiel zwei verschiedene Benutzeroberflächen über den gleichen Port betreiben.

Es sind je nach dem Typ der Anwendung ein bis vier Ports vorgesehen: ▶



Hexagonale Architektur mit Adaptern für alle Ports (Bild 4)



Clean Architecture im Überblick (Bild 5)

- Ereignisquellen wie Benutzerschnittstellen
- Persistenz oder Datenbank
- Benachrichtigungen
- Verwaltung zur Steuerung der Komponente

Eine hexagonale Architektur mit Adaptern für sämtliche Ports ist in Bild 4 zu sehen.

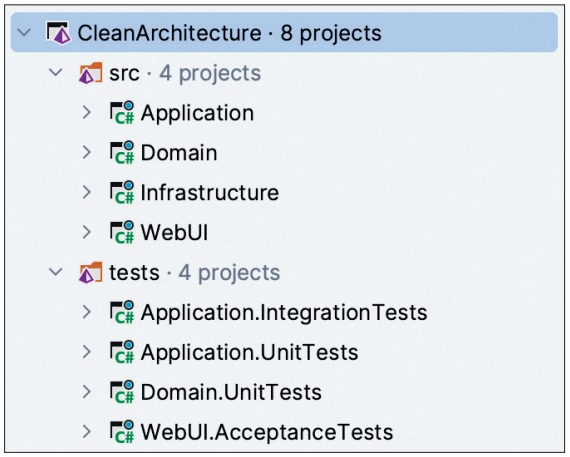
Drehen wir uns im Kreis?

Mit dem Ziel, für Attribute wie die Unabhängigkeit von Frameworks, Datenbanken, UIs sowie Testbarkeit der bisherigen Architekturen eine konkret umsetzbare Implementierung bereitzustellen, ist 2012 Rob C. Martin mit der „Clean Architecture" angetreten.

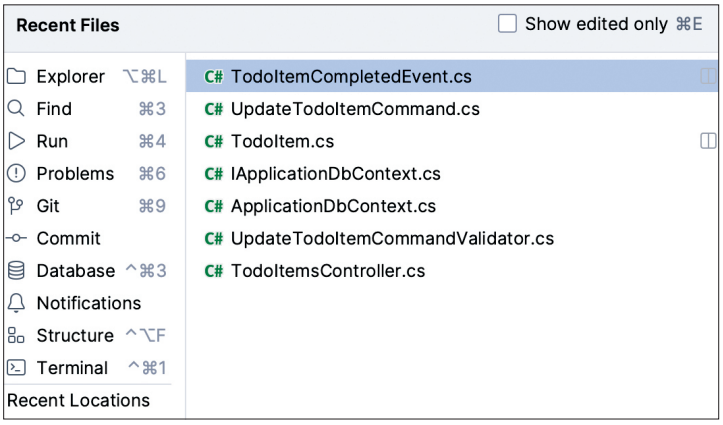
Die wichtigste Regel dieser Architektur (Bild 5) lautet, dass Abhängigkeiten nur von den äußeren zu den inneren Ringen der Software zeigen können. Code innerhalb eines Rings kennt Code eines Rings, der ihn umschließt, nicht. Dies gilt nicht nur für Code, sondern auch für Datenformate wie zum Beispiel DTOs oder Datenbankobjekte.

Im innersten Kreis liegen die Entitäten, die keinerlei Abhängigkeiten besitzen. Im Ring *Use Cases* wird aus den Entitäten und Geschäftsregeln die Fachlichkeit abgebildet. Der dritte Ring bildet Schnittstellen zur Außenwelt ab, das heißt für UIs, Datenbanken und dergleichen. Die Implementierung der Schnittstellen erfolgt im äußersten Ring.

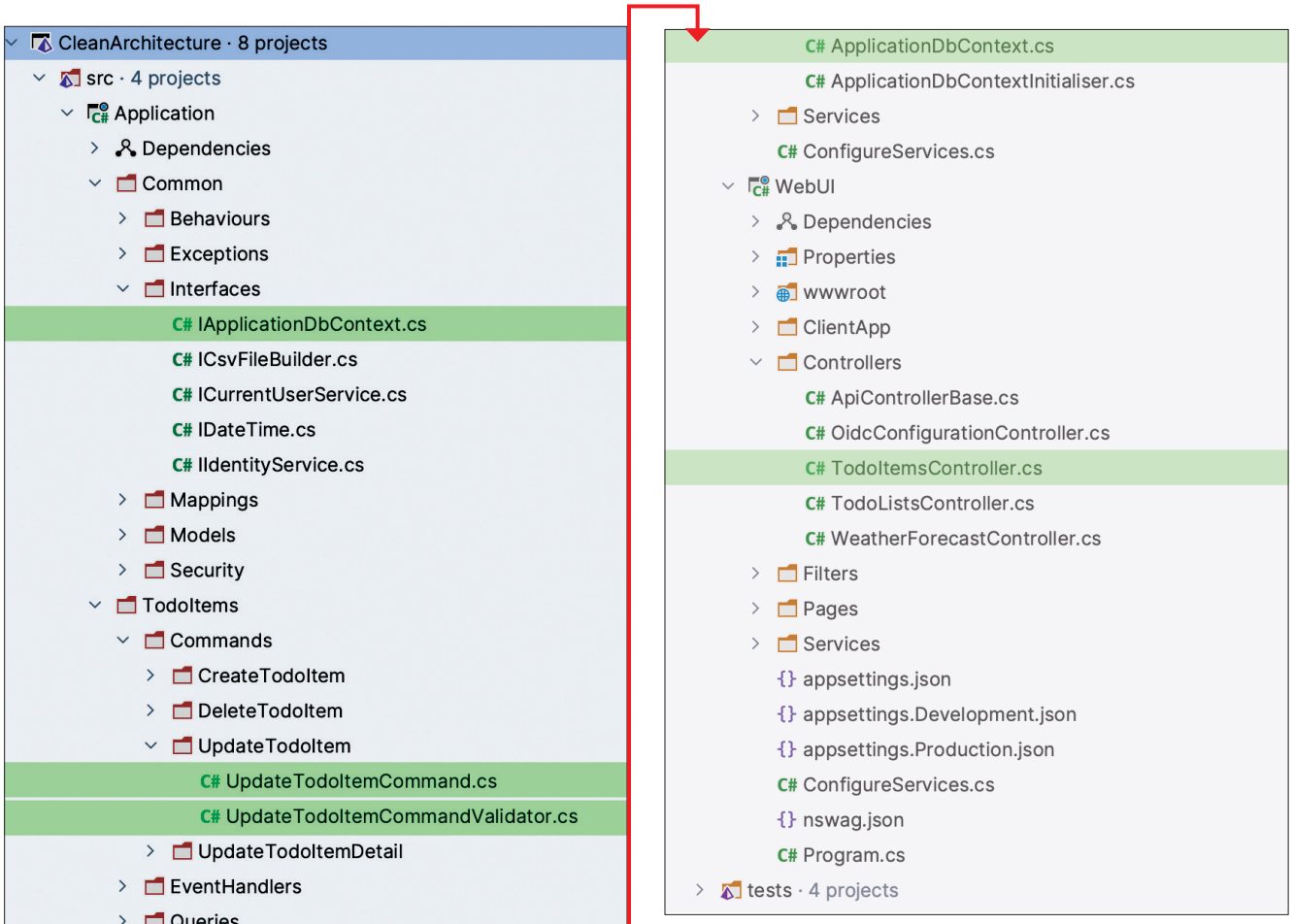
Abgesehen von der Verwendung von Kreisen und der Abhängigkeitsregel (nur von außen nach innen) ist dieser Architekturstil letztlich sehr ähnlich zur hexagonalen Architektur. Sie stellt außerdem auch eine Implementierung der Onion-Architektur von Jeffrey Palermo aus dem Jahr 2008 dar. Aufgrund der Tatsache, dass diese Architektur sehr konkret be-



Clean Architecture für C# (Bild 6)



Dateien, in denen sich die Implementierung des Update-Kommandos verbirgt (Bild 7)



Solution-Explorer-Darstellung der Dateien, in denen sich die Implementierung des Update-Kommandos verbirgt (Bild 8)

schrieben ist und es auch Templates gibt, hat sie inzwischen einen hohen Verbreitungsgrad erreicht.

Alles gut?

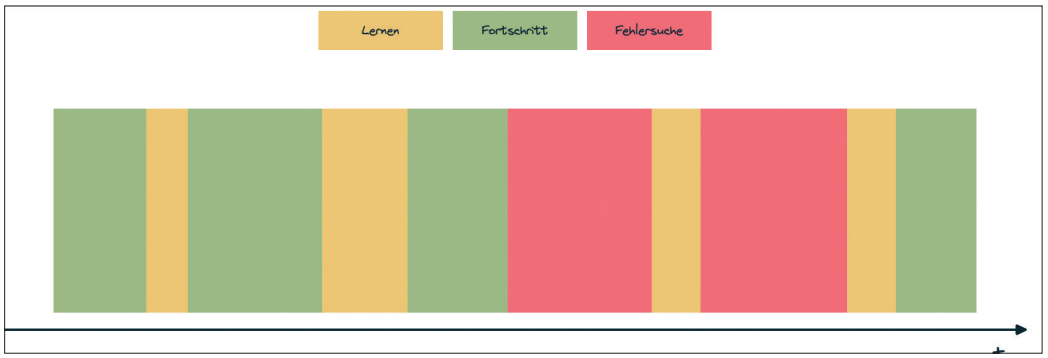
Nachdem wir uns die gängigen Architekturstile angesehen haben, nehmen wir nun nochmals Bezug auf den Beginn des Artikels und stellen uns die Frage, ob diese die Probleme von Softwareentwicklern tatsächlich lösen.

Allen Stilen gemein ist die Tatsache, dass sie sowohl Zuständigkeiten klar abgrenzen als auch die Richtung der Abhängigkeiten ausdrücklich definieren.

Warum treten dann aber sowohl bei monolithischen als auch bei im Microservices-Stil implementierten Varianten dieser Architekturen noch immer Probleme auf, die letztlich wieder auf ein Durcheinander im Code zurückzuführen sind?

Betrachten wir dazu den Vorgang, wie Entwickler sich der Aufgabenstellung zur Implementierung eines Features, von Bug Fixes oder dem Durchführen eines Refactorings in einer bestehenden Code-Basis nähern.

Ist die Aufgabe fachlich verstanden, suchen wir im Code nach einem Einstiegspunkt, um mit der Umsetzung zu beginnen. Je nach dem Grad der Vertrautheit mit dem Code und den zur Verfügung stehenden Werkzeugen ist das ein mehr oder minder komplexes Unterfangen. Aber auch der Code selbst spielt eine Rolle. ▶



Software entwickeln als Histogramm (Bild 9)

Als Basis unserer Analyse soll uns eine relativ populäre Vorlage zur Clean Architecture für C# dienen, die auf GitHub zu finden ist [1] und derzeit mit über 10 000 Sternen bewertet ist. Nach dem Öffnen präsentiert sie sich mit vier Projekten sowie den zugehörigen Testprojekten, die der Konvention von Clean Architecture folgen (siehe Bild 6).

Die Beispielanwendung dient der Verwaltung von Aufgaben, das heißt dem Hinzufügen, Ändern, Entfernen und Erledigen – eine typische To-do-App also.

Möchte man nun einen Use-Case innerhalb der Anwendung vollständig nachvollziehen, gilt es, alle relevanten Stellen im Code zu identifizieren. Ohne UI-Komponenten sind es im Fall des Kommandos zum Ändern oder Erledigen einer Aufgabe in unserer Beispielanwendung sieben Dateien, die man potenziell geöffnet hat, um eine Änderung am Code durchgängig zu implementieren. Von dieser Zahl ebenfalls ausgenommen sind Basisklassen, die Teil der Infrastruktur der Basisanwendung oder verwendeter Frameworks sind – aufgelistet in Bild 7.

Bedingt durch die Architektur befinden sich diese Dateien allerdings nicht so wie in Bild 7 nahe beieinander. Vielmehr sind sie über die vier bereits genannten Projekte verteilt. Somit entspricht die tatsächliche Situation der Darstellung in Bild 8 – die relevanten Dateien sind grün hervorgehoben.

Würde man die Tätigkeiten eines Entwicklers beim Erledigen einer Aufgabe aufzeichnen, so ergäbe sich ein Histogramm aus Blöcken von „Lernen/Verstehen“, „Fortschritt erzielen“ und „Fehlersuche“ (Bild 9).

Idealerweise würde der Entwicklertag nur aus „Fortschritt erzielen“ bestehen. Zumindest das Erlernen der Fachlichkeit lässt sich aber nicht ausschließen. Je nach persönlichem Kenntnisstand von Tools und Programmiersprache ist auch hier ein Lernprozess notwendig. Doch selbst wenn ein Entwickler sowohl von Sprache/Technologie als auch der Domäne ein gutes Verständnis hat, müssen beide im Code noch zusammenfinden.

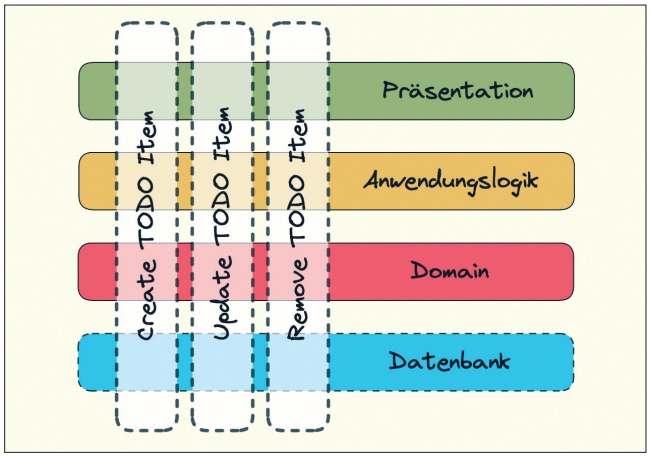
Weshalb ist es nun trotz etablierter Architekturmuster und zugehöriger Vorlagen so schwierig, den Durch- und Überblick zu wahren und langfristige Wartbarkeit zu gewährleisten? Eine Ursache hierfür ist, dass alle drei vorgenannten Ansätze den Code ausschließlich nach technischen Aspekten strukturieren und entkoppeln (Controller zu Controllern, Command zu Commands et cetera).

Ein wichtiger Aspekt bei der Lesbarkeit und damit in der Folge auch bei der Wartbarkeit ist jedoch auch, wie nahe Code, der eine Aufgabe erfüllt, beieinander liegt. Dieser Aspekt nennt sich Kohäsion und steht im Widerspruch zu loser Kopplung. Im Fall der genannten Architekturstile ist die Kohäsion – insbesondere die funktionale – sehr gering. Diese fehlende Kohäsion macht es schwierig, ein Abbild des gesamten Prozesses zur Lösung einer Aufgabe im Kopf des Entwicklers abzubilden.

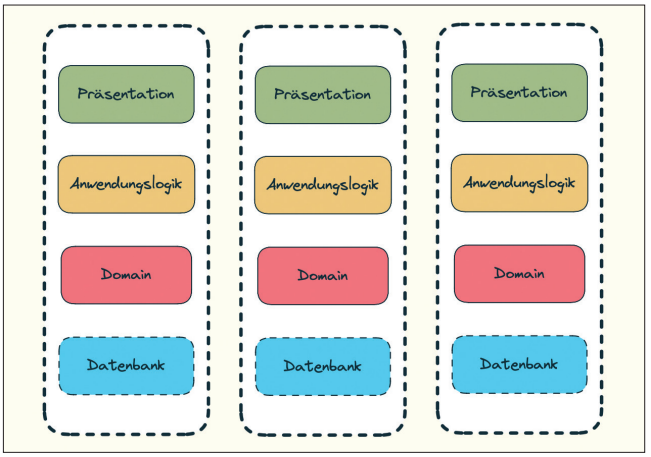
Wie lassen sich nun lose Kopplung und Kohäsion in einer Code-Basis gemeinsam abbilden?

Es wächst zusammen, was zusammengehört

Die Lösung heißt Vertical Slice Architecture, kurz VSA – entwickelt 2018 von Jimmy Bogard [2]. Der Begriff „Slice“ steht



Features umgesetzt in Clean Architecture (Bild 10)



Features umgesetzt als vertikale Schnitte mit VSA (Bild 11)

hierbei für einen Ausschnitt der Software in vertikaler Richtung, bezogen auf die bisherigen horizontalen Layer. Dies steht im Gegensatz zum bisherigen Ansatz der Clean Architecture, bei dem sich die Features nicht mehr über die Layer erstrecken (Bild 10): In VSA sind die Layer in den Features enthalten, wie in Bild 11 dargestellt.

Wie aus Bild 10 ersichtlich ist, sind die Features nun in sich abgeschlossen und beinhalten die bisherigen horizontalen, technischen Layer – je Feature. Dabei ist ein Feature im Sinne der VSA zum Beispiel das Anlegen einer neuen Aufgabe in unserer Beispielanwendung. Eine Datenabfrage (Query) ist ebenfalls ein eigenständiges Feature.

Die wichtigste Frage, die sich nun stellt: Wie sieht unser Code jetzt aus? Bild 12 zeigt eine mögliche Variante. Die Anwendung ist nun ein einziges ASP.NET-MVC-Projekt, das einen Ordner *Features* enthält. Dieser beinhaltet je Feature einen weiteren Unterordner, in welchem dann die bereits bekannten Dateien enthalten sind.

Beim Controller ergibt sich jedoch eine Änderung der Zuständigkeit: Er ist nun nicht mehr für alle Methoden der To-do-Items zuständig, sondern nur noch für zum Beispiel *UpdateTodoItem*. Entsprechend ändert sich der Name. Listing 1 zeigt diese Art der Implementierung.

Eine weitere Spielart – die der Autor bevorzugt – ist, dass alle Klassen, die zu einem Feature gehören, in einer einzigen Datei abgelegt werden. Der Name der Datei lautet dann wie das Feature, zum Beispiel *UpdateTodoItem.cs*. Dieses Vorgehen, zusammengehörigen Code in eine Datei zu speichern, ist übrigens in der F#-Welt gängige Praxis.

Für die Anordnung der Klassen in der Datei bieten sich zwei sinnvolle Varianten an: Entweder man beginnt mit dem Controller, der die äußere Schnittstelle des Features liefert, und dann folgen alle Abhängigkeiten des Controllers sowie deren Abhängigkeiten.

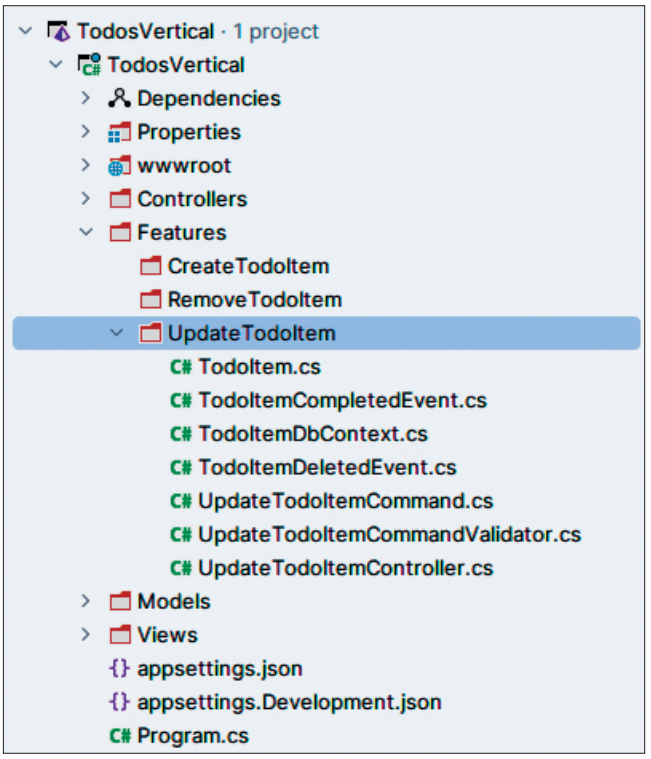
Die zweite Variante orientiert sich an der F#-Praxis und sortiert beginnend mit der Abhängigkeit der niedrigsten Ebene und damit dem Controller am Ende der Datei. Leider erlaubt C# es, Abhängigkeiten in einer Datei unterhalb deren Verwendung zu implementieren.

Der Ansatz, alle relevanten Dateien eines Features in einen Ordner oder gar eine Datei zu packen, endet übrigens nicht bei reinem C#-Code. Auch ASP.NET Core Razor Views können direkt im Feature-Ordner enthalten sein (Bild 13).

Damit Tools und Compiler mit dieser Erweiterung klarkommen, sind je nach Tool folgende Anpassungen notwendig: Um dem C#/ASP.NET-Compiler den veränderten Pfad zur View mitzuteilen, implementiert man einen sogenannten View Location Expander. Dessen Implementierung ist trivial und enthält die Liste der Pfade, die nach CSHTML-Dateien durchsucht werden sollen. Listing 2 zeigt eine solche Implementierung entsprechend unserer bisher verwendeten VSA-Code-Struktur.

Der Platzhalter *{0}* steht hierbei für den Namen der Aktion und somit View, während der Platzhalter *{1}* für den Namen des Controllers steht.

Der View Location Expander muss nun noch in der Service-Registrierung bekannt gemacht werden:



VSA mit einer Datei je Klasse (Bild 12)

```
builder.Services.AddControllersWithViews();

// Registrierung des View Location Expander
builder.Services.Configure<RazorViewEngineOptions>(
  o => o.ViewLocationExpanders.Add(
  new FeatureFolderLocationExpander())
);
```

Verwendet man Visual Studio zusammen mit JetBrains ReSharper oder gar Rider als IDE, stehen deutlich mehr Möglichkeiten zur Navigation im Code zur Verfügung. Dies schließt auch Razor Views ein. Damit beide Tools die veränderten Pfade berücksichtigen können, stellt JetBrains ein NuGet-Paket namens *JetBrains.Annotations* bereit. Dort sind Attribute enthalten, über die ReSharper beziehungsweise Rider mitgeteilt werden kann, in welchen Ordnern und Dateien Views enthalten sind.

Innerhalb einer *AssemblyInfo.cs*-Datei lässt sich sowohl der Speicherort für normale Views als auch der für Partial Views anpassen. Es stehen weitere Attribute für zum Beispiel Razor Areas und anderes zur Verfügung.

```
[assembly: AspMvcViewLocationFormat(
  @"~/Features/{1}/{0}.cshtml")]
[assembly: AspMvcPartialViewLocationFormat(
  @"~/Features/{1}/{0}.cshtml")]
```

Wohin geht die Reise?

Wenden wir uns nach dem Code nun nochmals den theoretischen und organisatorischen Aspekten der neuen Architektur zu. Je nach Spielart – mehrere oder eine Datei je Fea- ▶

ture in einem Ordner – hat sich die Kohäsion mehr oder weniger deutlich verbessert. Gleichzeitig mussten wir die bisher lose Kopplung nicht aufgeben. Unsere Situation hat sich dadurch also verbessert und die Navigation im Code ist deutlich einfacher geworden. Neben den angestrebten Zielen ergeben sich aber weitere Möglichkeiten, die nun offenstehen.

Werfen wir nochmals einen Blick auf Bild 10, sehen wir, dass alle Features nach dem gleichen Schema mit den gleichen Schichten implementiert sind.

Dies ist aber nun nicht mehr zwingend notwendig. Vielmehr können wir dem jeweiligen Kontext Beachtung schenken, in dem das Feature entsteht. Implementieren wir zum Beispiel ein Feature aus der Kerndomäne unserer Anwendung, möchten wir dieses vielleicht mit Event Sourcing umsetzen und die Vorteile dieses Ansatzes nutzen. Ein anderes Feature in der gleichen Anwendung ist aber möglicherweise nicht Teil der Kerndomäne, sondern nur aus einer unterstützenden oder gar einer generischen Domäne. Hier möchten

Listing 1: VSA-Implementierung des To-do-Beispiels

```
using MediatR;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using TodosVertical.Common;

namespace TodosVertical.Features.UpdateTodoItem;

[Authorize]
public class TodoItemsController : ApiControllerBase
{
  [HttpPut("{id}")]
  public async Task<ActionResult> Update(
    int id,
    UpdateTodoItemCommand command
  )
  {
    if (id != command.Id)
    {
      return BadRequest();
    }

    await Mediator.Send(command);

    return NoContent();
  }
}

public record UpdateTodoItemCommand : IRequest
{
  public int Id { get; init; }

  public string? Title { get; init; }

  public bool Done { get; init; }
}

public class UpdateTodoItemCommandHandler :
    IRequestHandler<UpdateTodoItemCommand>
{
  private readonly IApplicationDbContext _context;

  public UpdateTodoItemCommandHandler(
    IApplicationDbContext context
  )
  {
    _context = context;
  }

  public async Task<Unit> Handle(
    UpdateTodoItemCommand request,
    CancellationToken cancellationToken
  )
  {
    var entity = await _context.TodoItems
      .FindAsync(new object[] { request.Id },
      cancellationToken);

    if (entity == null)
    {
      throw new NotFoundException(nameof(TodoItem),
        request.Id);
    }

    entity.Title = request.Title;
    entity.Done = request.Done;

    await _context.SaveChangesAsync(
      cancellationToken);

    return Unit.Value;
  }
}

public class TodoItemCompletedEvent : BaseEvent
{
  public TodoItemCompletedEvent(
    TodoItem item
  )
  {
    Item = item;
  }

  public TodoItem Item { get; }
}
```

Listing 2: VSA-Code-Struktur mit Suchpfaden für die View

```
public class FeatureFolderLocationExpander :
  IViewLocationExpander
{
  public void PopulateValues(
    ViewLocationExpanderContext context
  )
  {
  }

  public IEnumerable<string> ExpandViewLocations(
    ViewLocationExpanderContext context,
    IEnumerable<string> viewLocations
  )
  {
    return new[]
    {
      "~/Features/{1}/{0}.cshtml"
    };
  }
}
```

wir den Aufwand gering halten und setzen diesen einfach mit Entity Framework als Persistenzschicht um. Außerdem beinhaltet das Kernfeature vielleicht ein HTTP-API, während die unterstützende Domäne nur ein Web-Frontend bereitstellt. Möglicherweise genügt auch ein Transaction Script für die Lösung der Aufgabenstellung.

Wie bereits erwähnt, werden Datenabfragen ebenfalls als Features betrachtet und implementiert. Damit haben wir also eine Trennung zwischen Kommandos und Abfragen. Dem ein oder anderen Leser dürfte dieses Muster bekannt vorkommen: Wir haben damit CQRS in seiner einfachsten Form implementiert.

Ein Aspekt, den wir bei der Betrachtung der Architekturen eingebracht haben, war die Sicht auf den Entwickleralltag. Hat sich dieser durch die Vertical Slice Architecture auch verändert?

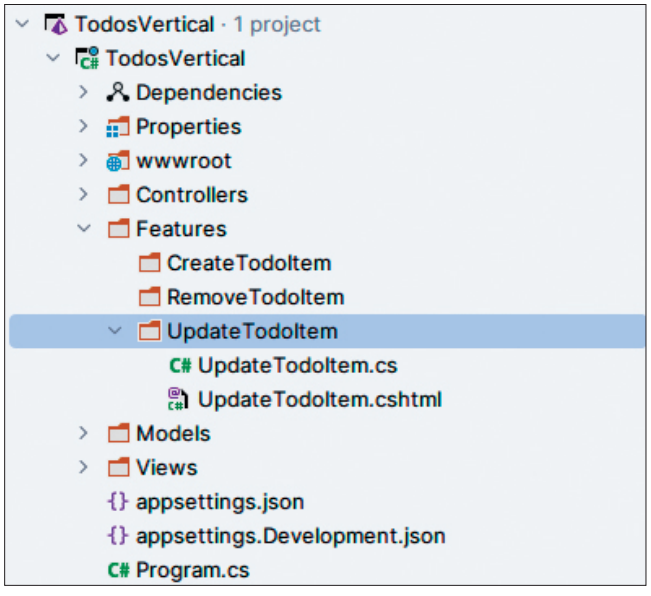
Aus der eigenen Praxis mit VSA kann der Autor bestätigen, dass die Phasen von Lernen und Fehlersuche im Alltag durchaus kürzer werden. Auch die kognitive Belastung hat sich reduziert, da man Code jetzt in seiner Gesamtheit einfacher und vollständiger erfassen kann. Daraus resultiert auch, dass die Modellierung von Erweiterungen oder Änderungen des bestehenden Codes leichter fallen. Selbstredend ist die Navigation im Code selbst um Größenordnungen einfacher, da man fast immer innerhalb der gleichen Datei oder wenigstens innerhalb des gleichen Ordners navigiert (je nach Variante).

Ein weiterer positiver Nebeneffekt, der sich einstellt: Das Potenzial von Merge-Konflikten in der Quellcode-Verwaltung reduziert sich. Die Wahrscheinlichkeit, am gleichen Feature zu arbeiten, ist nun deutlich geringer, da sich der Umfang reduziert hat.

Alle diese Änderungen im Alltag führen auch dazu, dass notwendige Änderungen am Code nicht mehr aufgeschoben werden, aus der Angst heraus, Code eines anderen Features zu beschädigen. Die Agilität verbessert sich also ebenfalls.

Da der Code nun bereits nach Features beziehungsweise Kontext strukturiert ist, fällt auch Extrahieren eines solchen Blocks als eigenständiger Service viel einfacher.

Zusammenfassend lässt sich also sagen, dass mit der Vertical Slice Architecture die Organisation des Codes nun auch fachliche Belange berücksichtigt. Ausgehend von einer Clean-Architecture-Implementierung lässt sich mit geringem Aufwand stufenweise ein Refactoring durchführen, an dessen Ende eine Vielzahl an weiteren Optionen für die Architektur der Software ermöglicht werden. ■



VSA umgesetzt mit Razor Views (Bild 13)

[1] Vorlage zur Clean Architecture für C# auf GitHub, www.dotnetpro.de/SL2310FeatureSlices1

[2] Jimmy Bogard, Vertical Slice Architecture bei YouTube, www.dotnetpro.de/SL2310FeatureSlices2

Alexander Zeitler
lebt und arbeitet als Cloud Solutions Architect in Karlsruhe. Er entwickelt B2B-Plattformen im industriellen Umfeld mit den Schwerpunkten AWS, .NET und Node.js.
alexander.zeitler@pdmlab.com

dnpCode A2310FeatureSlices