DIE IODA ARCHITEKTUR IM VERGLEICH

RALF WESTPHAL

dotnetpro

Die IODA Architektur im Vergleich

Ralf Westphal und dotnetpro

Dieses Buch wird verkauft unter
http://leanpub.com/ioda-architektur-im-vergleich-dnp

Diese Version wurde veröffentlicht am 2020-12-27

Ebenfalls von Ralf Westphal

Test-first Codierung

Softwareentwurf mit Flow-Design

Software Anforderungsanalyse mit Slicing

Inhaltsverzeichnis

Vorwort

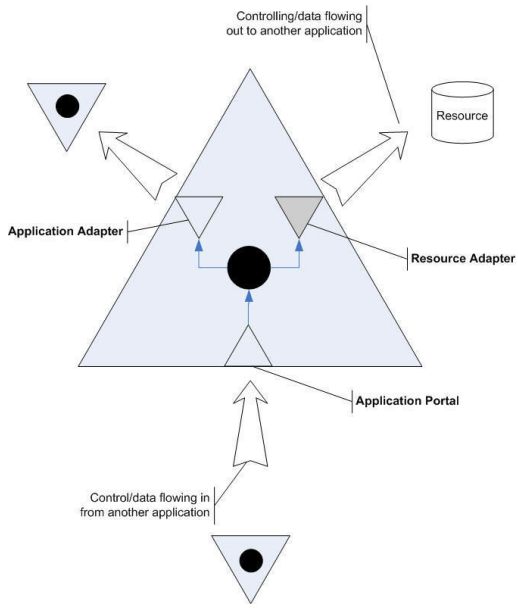
Seit Ende der 1990er befasse ich mich mit Softwarearchitektur explizit. Vorher hatte ich Software einfach "irgendwie zusammengeschraubt", glaube ich. Das hatte genügend gut geklappt, so dass ich damit Geld verdienen konnte; die Kunden der Firma, die ich mit meinem Geschäftspartner hatte, waren zufrieden mit unserer Software.

Doch dann irgendwann war das nicht mehr genug. In die Microsoft-Bubble, in der ich mich damals befand, drang etwas Neues ein. Entwurfsmuster für die Objektorientierung waren angesagt und dann kam sogar Microsoft als Technologiehersteller mit Empfehlungen für die Strukturierung von Software um die Ecke. Ich erinnere mich ans *Schichtenmodell*, dann an *N-Tier Architecture*, dann an *Emissaries and Executants* (ich glaube, so hieß das eine Zeit lang)… Auch wenn die Details verschwimmen, eines habe ich noch im Gefühl: Softwarearchitektur war wichtig geworden.

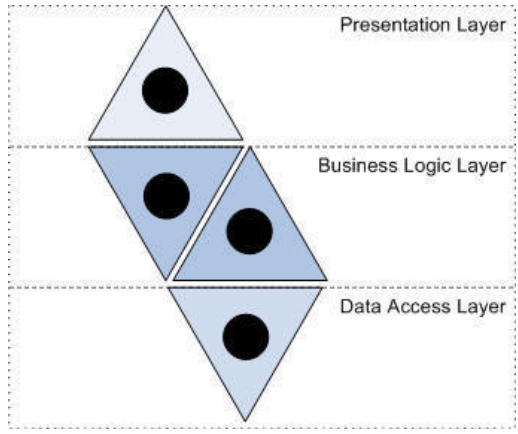
Wie es dann so kam, hat mich das Thema nicht wieder verlassen. Ich war vom Technologieanwender zu einem "Planer" von Technologieanwendung geworden.

Allerdings konnte ich schon bald nicht mehr einfach akzeptieren, was an Architekturmustern empfohlen wurde. Immer fehlte irgendetwas oder kam mir nicht plausibel vor. Anfang 2005 habe ich dieses Gefühl dann so ernst genommen, dass ich anfing, an einem eigenen Architekturmodell zu tüfteln. Die Softwarezelle ward geboren. Hier zwei Bilder aus dieser Zeit, mit denen ich das Konzept in meinem damaligen Blog erklärt und entwickelt habe:[1]

[1] Interessanterweise war ich nicht der einzige, dem da etwas fehlte. Später habe ich erfahren, dass Alistair Cockburn zur selben Zeit an seiner Hexgonal Architecture gearbeitet hat. Es lag da also etwas in der Luft…



Eine frühe Form der Softwarezelle aus dem April 2005



Softwarezellen im Verbund für eine verteilte Architektur

Damals war mir sehr wichtig, die Geschäftslogik in den Mittelpunkt zu rücken. Sie schien mir in anderen Architekturmustern zu wenig betont

und gerade für verteilte Anwendungen stiefmütterlich behandelt.[2]

Außerdem fand ich die ganze Herangehensweise an die Strukturierung von Software zu technisch, zu mechanisch. Wenn Software entwickelt wird, sich also entwickelt, über lange Zeit entwickelt, geradezu eine Evolution durchläuft… dann, so war mein Gedanke, sollte sie durch ein organischeres Bild beschrieben werden. Deshalb der Begriff *Softwarezelle*. Mit ihr, aus ihr wollte ich Software wachsen sehen.

Vielleicht entstand damals mein Interesse für nachhaltige Softwareentwicklung, das später zur Mitgründung der Clean Code Developer Initiative geführt hat. Mein Empfinden war einfach, dass viele Entwickler sich redlich bemühten, das eine oder andere Architekturmuster anzuwenden, um nicht zu schnell in die "Unwartbarkeit" zu laufen. Doch dieses Bemühen war zu selten von Erfolg gekrönt. Die Anwendung der Muster funktionierte nicht wie gewünscht, was immer wieder zu Frust geführt hat und der wiederum zu einer Hoffnung, dass Technologie das Dilemma doch bitte lösen möge.

Doch Technologie nimmt uns in der Softwareentwicklung das Nachdenken nicht ab - außer vielleicht in Sonderfällen. Wir müssen weiterhin verstehen und entscheiden. Und für das Entscheiden brauchen wir Heuristiken, Prinzipien, Konzepte.

Seitdem hat mich das Thema Softwarearchitektur also nicht losgelassen. Bei aller trivialen Korrektheit des Beraterspruchs "Es kommt darauf an…" glaube ich, dass es einen Rahmen gibt, in dem sich Softwarearchitektur bewegen sollte. Die konkrete Architektur eines Softwaresystems orientiert sich nur daran, sie prägt ihn individuell im Hinblick auf die nichtfunktionalen Anforderungen aus. Dabei kommt es natürlich darauf an, *wie* man das tut.

Doch es kommt eben nicht darauf an, *dass* man es tut. Softwarearchitektur ausgehend von Prinzipien und Mustern nicht explizit zu betreiben, halte ich für keine Option.

Aber welche Prinzipien und Muster? Darüber habe ich lange nachgedacht. Die Beschäftigung mit dem Clean Code Development hat mir dabei ge-

[2] In Abbildung 31 finden Sie die Softwarezelle auch heute noch wieder, selbst wenn ich sie in der Artikelreihe nicht so genannt habe. Ich wollte das Neue der IODA Architektur nicht noch mit einem solchen Begriff überladen, allemal, da ich auf die Konsequenzen für die Verteilung nicht eingegangen bin.

holfen. Das eine hat das andere befruchtet. Deshalb spreche ich heute auch weniger von Clean Code; meine Trainings laufen unter einer anderen Überschrift, um den Bogen weit genug spannen zu können. Denn worum geht es? Um langfristig hohe Produktivität.

Softwarearchitektur ist ein Aspekt des Wunsches, Software über möglichst lange Zeit möglichst anpassungsfähig (wandelbar) zu halten. Andere Aspekte gehören auch dazu: konsequente test-first Codierung, inkrementelle Anforderungsanalyse und Umsetzung, Zurückhaltung bei der Veränderung von Produktionscode, Verzicht auf die Aufwandsschätzung zugunsten von Vorhersagen usw.

Clean Code hat Appeal für Entwickler, nicht für Manager. Es ist damit ein zu techschnischer Begriff für das, worum es geht. *Programming with Ease* hingegen spannt für mich einen Bogen, der einerseits weit genug ist und andererseits spezifisch genug. Ich möchte die Programmierung rundum erleichtern. Dazu gehört auch, die grobe Strukturierung von Code. Denn wer keine grundlegende Vorstellung von der Anatomie von Software hat, von ihren wiederkehrenden Funktionsbausteinkategorien und deren Zusammenhänge, der tut sich von Anfang an schwer mit jedem Softwareprojekt. Und dabei geht es noch nicht einmal um die Erfüllung spezieller nicht-funktionaler Anforderungen. Nein, es reicht schlicht Testbarkeit und Wandelbarkeit auch bei einem Monolithen, d.h. nicht verteilter Software, hoch halten zu wollen. Das ist Problem genug, um stets nach besseren Ansätzen zu suchen.

Das habe ich getan und tue es noch mit der IODA Architektur. Mit ihr stelle ich das, was mit Softwarezellen begonnen hat, auf ein Prinzipienfundament. Die Hexagonal Architecture und die Clean Architecture basieren auf den Prinzipien DIP/IoC. Die IODA Architektur basiert auf IOSP und PoMO als Ergebnisse einer Analyse der ursprünglichen Objektorientierung, wie sie Alan Kay 1968 gedacht hatte.[3]

Ich habe nichts gegen DIP/IoC. Im Gegenteil! Aber für mich ist da eben nicht Schluss. Für testbarere und wandelbarere Software brauchen wir ein Architekturmuster, das darüber hinaus geht.

Mir scheint, dass ich 2015 das erste Mal den Begriff IODA Architektur in

[3] Für eine ausführliche Herleitung siehe meine Artikelserie OOP as if you meant it bzw. den Band Softwareentwurf mit Flow-Design aus meiner *Programming with Ease* Reihe.

einem Blogartikel[4] benutzt hatte. 2018 habe ich den aktuellen Stand dazu dann in drei Artikeln in der dotnetpro zusammengefasst. Diese Artikel sind in diesem kleinen Buch zusammengefasst, um die Lektüre einfacher und unabhängig von einem Abonnement der dotnetpro zu machen. Vielen Dank an Chefrefakteur Tilman Börner und die Ebner Media Group, eine solche herausgelöste Veröffentlichung zu ermöglichen. Natürlich habe ich diese Gelegenheit genutzt, die Artikel durchzusehen, hier und da zu ergänzen und am Ende noch ein Update hinzuzufügen, das einzuarbeiten schwierig gewesen wäre. Ich hoffe, auf diese Weise diese Perspektive auf Softwarearchitekturmuster einem größeren Interessentenkreis zugänglich zu machen.

Viel Spaß bei der Lektüre!

-Ralf Westphal, Bansko/Bulgarien im Dezember 2020

[4] http://geekswithblogs.net/theArchitectsNapkin/archive/2015/04/29/the-ioda-architecture.aspx

1 - Eine Kritik bisheriger Architekturmodelle

Ich kann sie nicht mehr hören die Anpreisungen des Architekturmusters „Schichtenmodell“. In der dotnetpro wie anderswo spukt es immer wieder als klassische und deshalb gute Organisation von Code herum. Mir scheint das inzwischen ein Fall von Cargo Kult[5]: Irgendwer hat irgendwann seinen Code so strukturiert und damit einen Vorteil erlangt – und nun folgen dem Generationen von Entwicklern blind.

Was aber, wenn sich die Welt weitergedreht hat? Was, wenn man da etwas mechanisch tut, ohne wirklich konsequent über die ursprünglichen Beweggründe nachzudenken? Das Ergebnis ist dann immer gleich: Es entsteht Unwohlsein, die Dinge werden schwierig – doch man weiß nicht so recht, woher das kommt. Man macht doch alles richtig, oder? Eher wohl nicht; vielleicht muss man sich einfach noch mehr bemühen. Also die Anstrengungen verdoppeln, das Muster einzuhalten. Und so entsteht dann noch mehr Unwohlsein.

„Been there, done that, got the t-shirt“, kann ich dazu sagen. Einst war ich auch Anhänger des Schichtenmodells und anderer seiner mustergültigen Geschwister. Doch irgendwann habe ich für mich realisiert: der Schmerz ist größer als der Nutzen. Ich muss die Muster nicht besser anwenden, sondern einen anderen Weg suchen, meine Software zu strukturieren.

Worauf ist dann gestoßen bin, davon möchte ich Ihnen im Folgenden berichten. Es ist eine Geschichte der Erleichterung. Softwareentwicklung macht mir jetzt wieder Spaß. Ich kann mich viel mehr auf die Lösung der Probleme konzentrieren, weil die Struktur mich nicht mehr in ein hinderliches Korsett zwängt.

Aber der Reihe nach. Lassen Sie mich noch vor dem “Musterspuk” beginnen.

[5] https://en.wikipedia.org/wiki/Cargo_cult

Am Anfang war der Monolith

Hier ist eine Challenge:

> *Schreiben Sie ein Programm, das die Gesamtzahl der Worte sowie die Zahl der verschiedenen Worte in einem Text unter Berücksichtigung einer Stoppwortliste bestimmt. Der Text wird entweder vom Benutzer eingegeben oder aus einer Datei gelesen, die der Benutzer bei Programmstart angibt.*

Das ist eine simple Aufgabe, denke ich. Dennoch ist da alles drin, was eine Software ausmacht: ein bisschen Benutzerschnittstelle, ein bisschen Fachlogik, ein bisschen Datenzugriff. Genug, um darauf das Schichtenmodell und andere Strukturierungsideen anzuwenden.

In diesem Beispiel geht es nicht darum, ein Technologiefeuerwerk abzubrennen. Eine Konsolenanwendung reicht völlig aus. Deren Anwendung könnte so aussehen:

```
$ wordcount.exe
Text eingeben: Es blaut die Nacht, die Sternlein blinken
6 Worte, davon 5 verschieden
$ wordcount.exe gedichtanfang.txt
6 Worte, davon 5 verschieden
$
```

Der eingegebene Text hat zwar 7 Worte, doch das Wort „es“ steht in der Datei mit den Stoppworten und wird deshalb nicht gezählt. Außerdem steht „die“ im Text zweimal, daher unterscheidet sich die Zahl der Worte von der der verschiedenen.

Sie können ja mal als Fingerübung selbst für die Challenge ein Programm schreiben. Beobachten Sie sich dabei: Wie gehen Sie vor? Wie strukturieren Sie den Code und warum?

Wenn Sie mitmachen und später vergleichen möchten, dann lesen Sie erstmal nicht weiter. Spoileralarm! Denn ich möchte Ihnen verschiedene „hoch entwickelte“ Lösungen vorstellen, um daran zu zeigen, warum das Schichtenmodell und Verwandte keine Option mehr für mich sind.

Aber zuerst eine Lösungsvariante, die gar nicht mehr geht. Oder ist sie eine, die womöglich noch häufig anzutreffen ist? Entscheiden Sie, ob Ihnen solcher Code wie in Abbildung 1 immer noch über den Weg läuft.

Den Code bezeichne ich als monolithisch: Nicht nur ist er nicht verteilt, er besteht auch nur aus Logik. Rekombinierbare Strukturelemente wie Funktionen oder Klassen sind vernachlässigbar. Der Code ist also quasi strukturlos aus architektonischer Sicht.

Die Anweisungen in der einzigen Funktion *Main()*, d.h. die Logik, tut zwar, was sie tun soll: das Programm ist funktional; doch verständlich oder testbar ist die Logik nicht.

```
internal class Program
{
    public static void Main(string[] args) {
        var stopwords = new string[0];
        if (File.Exists("stopwords.txt"))
            stopwords = File.ReadAllLines("stopwords.txt");

        var text = "";
        if (args.Length > 0)
            text = File.ReadAllText(args[0]);
        else {
            Console.Write("Text eingeben: ");
            text = Console.ReadLine();
            if (string.IsNullOrWhiteSpace(text)) return;
        }

        var candidateWords = text.Split(new[] {' ', '\t', '\n', '\r'},
                                        StringSplitOptions.RemoveEmptyEntries);
        var countTotal = 0;
        var countDistinct = 0;
        var stopwordsDirectory = new HashSet<string>(stopwords);
        var distinctWords = new HashSet<string>();
        foreach (var wortkandidat in candidateWords) {
            foreach(var m in Regex.Matches(wortkandidat, @"[\w\-]*"))
                if (!string.IsNullOrWhiteSpace(m.ToString())) {
                    var word = m.ToString();

                    if (!stopwordsDirectory.Contains(word)) {
                        countTotal++;

                        if (!distinctWords.Contains(word)) {
                            distinctWords.Add(word);
                            countDistinct++;
                        }
                    }
                }
        }

        Console.WriteLine($"{countTotal} Worte, davon {countDistinct} verschieden");
    }
}
```

Abbildung 1: Eine funktionale Lösung mit unwesentlicher Struktur

Sicher, das sind kaum 50 Zeilen. Die zu verstehen, sollte doch kein Problem sein. Warum sich hier mehr Aufwand mit mehr Struktur machen?

Erstens ist das hier ein Beispiel mit überschaubarer Funktionalität (und auch noch eigentlich geradliniger Logik), um eben Strukturierungsansätze zu demonstrieren. Selbst wenn das später ein bisschen künstlich und overengineert aussehen sollte, wird es hoffentlich die wesentlichen Punkte illustrieren helfen, um die es mir geht.

Zweitens glaube ich, dass wir viel sensibler sein sollten, was die Strukturierung angeht. Wir sollten uns nicht überschätzen in der Fähigkeit, Code zu verstehen. Die Zeit für einen Bugfix oder zum Einbau einer Erweiterung

ist immer knapp. Jede Minute, die wir beim Verstehen von Code sparen können, bevor wir ihn verändern, ist wichtig. Dafür aber müssen wir vorher, schon beim Schreiben etwas tun. Der Code-Autor muss an den späteren Code-Leser denken.

> *"Programs must be written for people to read, and only incidentally for machines to execute.", Harold Abelson & Gerald Jay Sussman*

Was macht den Code in Abbildung 1 aber so schwer zu verstehen? Es ist die kunterbunte Vermischung von Verantwortlichkeiten.

In Abbildung 2 habe ich die wesentlichen Verantwortlichkeiten farblich hervorgehoben. Sie sehen, das ist ein Flickenteppich. Verantwortlichkeiten sind verstreut über die Logik. Verantwortlichkeiten werden mit Kontrollstrukturen „geöffnet", um dann andere dazwischen zu schieben und sie erst später zu „schließen".

```
internal class Program
{
    public static void Main(string[] args) {
        var stopwords = new string[0];
        if (File.Exists("stopwords.txt"))
            stopwords = File.ReadAllLines("stopwords.txt");

        var text = "";
        if (args.Length > 0)
            text = File.ReadAllText(args[0]);
        else {
            Console.Write("Text eingeben: ");
            text = Console.ReadLine();
            if (string.IsNullOrWhiteSpace(text)) return;
        }

        var candidateWords = text.Split(new[] {' ', '\t', '\n', '\r'},
                                        StringSplitOptions.RemoveEmptyEntries);
        var countTotal = 0;
        var countDistinct = 0;
        var stopwordsDirectory = new HashSet<string>(stopwords);
        var distinctWords = new HashSet<string>();
        foreach (var wortkandidat in candidateWords) {
            foreach(var m in Regex.Matches(wortkandidat, @"[\w\-]*"))
                if (!string.IsNullOrWhiteSpace(m.ToString())) {
                    var word = m.ToString();

                    if (!stopwordsDirectory.Contains(word)) {
                        countTotal++;

                        if (!distinctWords.Contains(word)) {
                            distinctWords.Add(word);
                            countDistinct++;
                        }
                    }
                }
        }

        Console.WriteLine($"{countTotal} Worte, davon {countDistinct} verschieden");
    }
}
```

Abbildung 2: Unstrukturierter Code ist ein Flickenteppich aus Verantwortlichkeiten

So kann kein Fluss des Verständnisses entstehen. Der Code „erzählt keine Story“, in der etwas entlang einer Kausalkette passiert. Es fehlen Bedeutungseinheiten, die Sie auf einen Blick erfassen können. Alles müssen Sie sich durch Simulation der Ausführung der Anweisungen erst erschließen.

Das ist gruselig aufwändig und fehlerträchtig. Und ob der Code wirklich korrekt ist, lässt sich nicht ermitteln, ohne ihn manuell auszuführen. Keine der Verantwortlichkeiten kann gezielt mit automatisierten Tests überprüft werden. So lässt sich nicht zügig feststellen, ob der Code schon reif zur Auslieferung ist oder nach einer Änderung immer noch korrekt, also regressionsfrei.

So geht's nicht. Da sind wir uns einig, hoffe ich.

Aber das war die Situation zumindest früher auch nach Einführung der Strukturierten Programmierung. Vor diesem Hintergrund sind die ersten Architekturmuster entstanden. Zunächst Model-View-Controller (MVC)[6], dann das Schichtenmodell. Was für ein Sonnenaufgang über dem Monolithen.

Den Monolithen in Schichten spalten

MVC, Schichtenmodell, Hexagonale Architektur[7] und auch die Clean Architecture[8] verfolgen alle denselben Ansatz:

1. Sie definieren einen Kanon von Verantwortlichkeiten und verordnen die Spaltung der Logik nach diesen Verantwortlichkeiten in Module.
2. Sie geben genau vor, wie die Module in Beziehung stehen, d.h. einander kennen und nutzen dürfen.

Die Beliebtheit des Schichtenmodells scheint mir hierbei in einer Kombination aus leicht zu verstehenden Verantwortlichkeiten in angenehmer Granularität und sehr klaren Beziehungen zu bestehen (Abbildung 3).

[6] https://de.wikipedia.org/wiki/Model_View_Controller
[7] https://en.wikipedia.org/wiki/Hexagonal_architecture_%28software%29
[8] https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

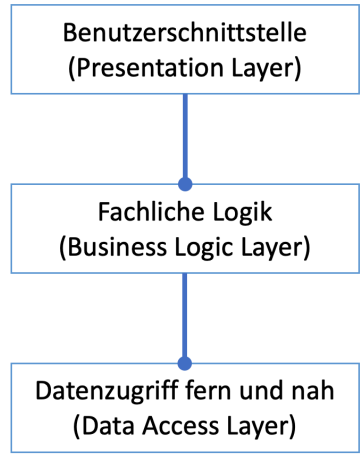


Abbildung 3: Das Schichtenmodell definiert verständliche Verantwortlichkeiten in klaren Beziehungen

Wenn ich dieses Muster auf den bisherigen Code zur Lösung der obigen Challenge anwende, dann ist damit tatsächlich etwas gewonnen: Verständlichkeit. Die Logik liegt nicht mehr auf einem Haufen, sondern ist verteilt auf Klassen als Module, so dass sich schon beim Betrachten des Projektes eine gewisse Übersicht einstellt (Abbildung 4).

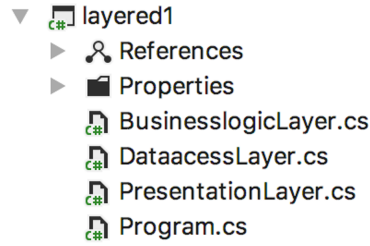


Abbildung 4: Schichten steigern die Übersichtlichkeit

Wer mit dem Schichtenmodell vertraut ist, sieht hier erstens Verantwortlichkeiten, unter denen er sich etwas vorstellen kann. Zweitens sind dem Betrachter sofort die grundsätzlichen Beziehungen klar. Das ist ja auch der Sinn der Einhaltung eines solchen Musters. Sie müssen kein Rad neu erfinden, sondern können sich darauf verlassen, dass Sie nichts grob falsch machen, wenn Sie nach dem Schema strukturieren. Mit einem Architek-

turmuster setzen Sie eine Brille auf, durch die Sie die Logik analysieren können; was Sie dabei identifizieren, stecken Sie in die kanonischen Module. Und Sie können auch noch annehmen, dass ein anderer Entwickler, der auch das Muster kennt, ihre Struktur versteht; er findet sich darin also von vornherein leicht(er) zurecht.

Die nächste Abbildung zeigt konkret die Schichtung der Klassen der Umsetzung aus Abbildung 4. Ordentlich, oder? Eine so organisierte Codebasis macht Freude. Alles hat seinen Platz. Da wissen Sie genau, wo was passiert.

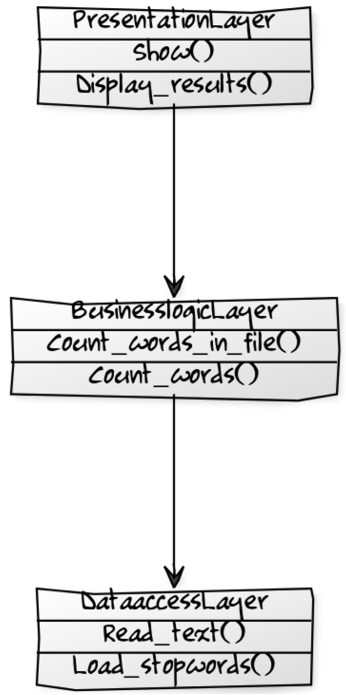


Abbildung 5: Konkrete Schichtung der Anwendungslogik

Ja, damit ist etwas gewonnen. Es ist besser als vorher, aber noch nicht wirklich gut. Denn schauen Sie sich einmal den Code der Businesslogik an:

```
class BusinesslogicLayer
{
    public static (int countTotal, int countDistinct) Count_words_in_file(string filename) {
        var text = DataacessLayer.Read_text(filename);
        return Count_words(text);
    }

    public static (int countTotal, int countDistinct) Count_words(string text) {
        var stopwordsDirectory = DataacessLayer.Load_stopwords();

        var candidateWords = text.Split(new[] {' ', '\t', '\n', '\r'},
                                        StringSplitOptions.RemoveEmptyEntries);
        var countTotal = 0;
        var countDistinct = 0;

        var distinctWords = new HashSet<string>();
        foreach (var wordCandidate in candidateWords) {
            foreach(var m in Regex.Matches(wordCandidate, @"[\w\-]*"))
                if (!string.IsNullOrWhiteSpace(m.ToString())) {
                    var word = m.ToString();

                    if (!stopwordsDirectory.Contains(word)) {
                        countTotal++;

                        if (!distinctWords.Contains(word)) {
                            distinctWords.Add(word);
                            countDistinct++;
                        }
                    }
                }
        }

        return (countTotal, countDistinct);
    }
}
```

Abbildung 6: Schlecht testbare Logik trotz Schichtung

Die groben Verantwortlichkeiten sind grundsätzlich hübsch getrennt, die Abhängigkeiten sind sauber ausgerichtet – doch gut testbar ist deshalb die eigentliche Businesslogik immer noch nicht. Denn die Businesslogik hängt immer noch von der Datenzugriffslogik ab. Es besteht eine *funktionale Abhängigkeit*: Logik in einer Methode ruft eine andere Methode auf, um zwischendurch deren Logik auszuführen.

Das hört sich normal an und findet sich bestimmt in Ihrem Code auch allerorten. Doch das macht es nicht besser. Solche *funktional abhängige Logik* ist schlicht nicht für sich allein testbar.

Natürlich ist die Logik in Abbildung 6 auch in anderer Hinsicht noch suboptimal. Doch das ist sekundär für den Punkt, um den es mir hier im Augenblick geht. Ich habe nur das minimal Nötige getan, um den monolithischen Code nach dem Schichtenmodell zu strukturieren. Das fundamentale Problem des Schichtenmodells geht nicht weg, wenn ich die Wortzählungslogik noch weiter refaktorisiere. Der Klumpen in *Count_words()* dient also der Unterstreichung des grundsätzlich zu lösenden Problems der funktionalen Abhängigkeiten.

Wer die Businesslogik testen will, der kann das im Moment trotz oder wegen Schichtenmodell nur tun, indem er ebenfalls die Logik der Datenzugriffsschicht ausführt. Das macht einen Businesslogik-Test langsamer und/oder umständlicher, weil eine Datei als Ressource bereitgestellt werden muss.

Nicht wirklich dramatisch in diesem trivialen Beispiel, doch wenn Sie sich das Szenario umfangreicher denken… dann kommt schon etwas zusammen an Overhead.

Wenn Abbildung 1 den Bewusstseinsstand in Sachen Anwendungsarchitektur bis Mitte der 1990er in vielen Projekten widerspiegelt, dann steht Abbildung 6 für den Ende der 1990er.

Schichten entkoppeln

In einer Co-Evolution mit bewussterer Anwendungsstrukturierung befand sich ab Ende der 1990er das Thema Testen. Die ersten Unit Testing Frameworks kamen auf.

Wo klare Verantwortlichkeiten in Modulen freigestellt waren, konnten überhaupt erst automatisierte Tests feingranular ansetzen. Aber um in einer sauberen Hierarchie automatisierte Tests punkgenau nur gewisse Logik testen lassen zu können, brauchte es Entkopplung der Verantwortlichkeiten.

Auftritt DIP: Mit dem *Dependency Inversion Principle*[9] wurde es möglich, Tests auf eine Schicht zu fokussieren.

Der Trick besteht darin, Compilezeitabhängigkeiten von Laufzeitabhängigkeiten zu trennen. Zur Compilezeit besteht nach DIP keine direkte funktionale Abhängigkeit von Logik einer Schicht zu Logik einer anderen. Eine obere Schicht hängt nicht von einer konkreten unteren ab, sondern lediglich von einer Abstraktion:

[9] https://de.wikipedia.org/wiki/Dependency-Inversion-Prinzip

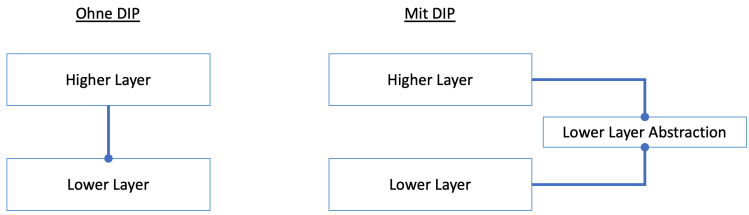


Abbildung 7: Mit dem DIP werden Schichten entkoppelt

Abstraktionen sind gewöhnlich Interfaces oder abstrakte Klassen. Die können von der niedrigen Schicht implementiert werden – aber man kann auch eine Attrappe für eine niedrige Schicht so aussehen lassen. Doch eins nach dem anderen.

Zuerst die Anwendung mit verbesserter Schichtenarchitektur in im Überblick. Hinzugekommen sind die Interfaces für die Module der bisherigen Schichten:

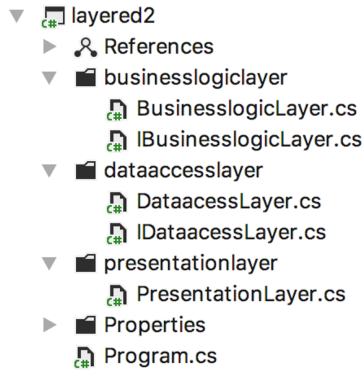


Abbildung 8: Eine Schichtenarchitektur mit DIP

Der geübte Softwerker lässt beim Anblick dieser Module sogleich vor seinem geistigen Auge ein Beziehungsgeflecht wie entstehen und weiß: alles hübsch entkoppelt und testbar.

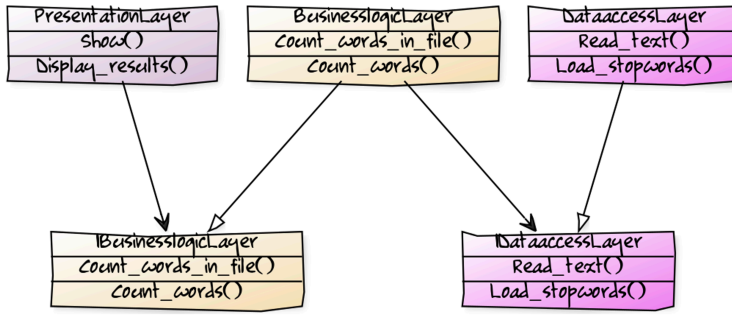


Abbildung 9: Über Interfaces entkoppelte Schichten

Und was ist der Nutzen des ganzen Aufwands? Im nächsten Codeausschnitt sehen Sie, wie nun mit einer Attrappe die Businesslogik unabhängig von darunterliegenden Details getestet werden kann. Die Implementation der aufrufenden Logik ist von der aufgerufenen durch das Interface entkoppelt; erst zur Laufzeit wird bestimmt, wer aufgerufen wird.

Statt einen Datenzugriff konkret zu durchlaufen, werden die Stoppworte im Test hart verdrahtet. Das ist trivial in puncto Laufzeit und Komplexität und einfacher, als eine Stoppwortdatei zu benutzen.

```
[TestFixture]
public class BusinesslogicLayer_tests
{
    [Test]
    public void Count_words()
    {
        var dal = new Mock<IDataacessLayer>();
        dal.Setup(x => x.Load_stopwords()).Returns(new HashSet<string>(new[] {"es"}));
        var sut = new BusinesslogicLayer(dal.Object);

        var (countTotal, countDistinct) = sut.Count_words("Es blaut die Nacht, die Sternlein blinken");

        Assert.AreEqual(6, countTotal);
        Assert.AreEqual(5, countDistinct);
    }
}
```

Abbildung 10: Mit einer Attrappe wird das Testen von abhängiger Logik einfach

Dass der Businesslogik aber überhaupt eine Attrappe untergeschoben werden kann, ist der Anwendung des *Inversion of Control (IoC)*[10] Prinzips geschuldet. Dessen Manifestation besteht hier im Hineinreichen der Laufzeitabhängigkeit in die Businesslogik durch ihren Konstruktor (*constructor injection).

[10] https://de.wikipedia.org/wiki/Inversion_of_Control

```
internal class Program
{
    public static void Main(string[] args) {
        var dal = new DataacessLayer();
        var bl = new BusinesslogicLayer(dal);
        var pl = new PresentationLayer(bl);

        pl.Show(args);
    }
}
```

```
class BusinesslogicLayer : IBusinesslogicLayer
{
    private readonly IDataacessLayer _dal;

    public BusinesslogicLayer(IDataacessLayer dal) { _dal = dal; }


    public (int countTotal, int countDistinct) Count_words_in_file(string filename) {
        var text = _dal.Read_text(filename);
        return Count_words(text);
    }

    public (int countTotal, int countDistinct) Count_words(string text) {
        var stopwordsDirectory = _dal.Load_stopwords();
```

Abbildung 11: Injizieren der konkreten Implementation einer abstrakten Abhängigkeit zur Laufzeit

Die Abhängigkeit vom Interface *IDataaccess* zur Compilezeit wird zur Laufzeit durch die Injektion einer Implementation des Interfaces befriedigt. Nachfolgende sind die Compilezeit- und Laufzeitabhängigkeiten zusammen visualisiert.

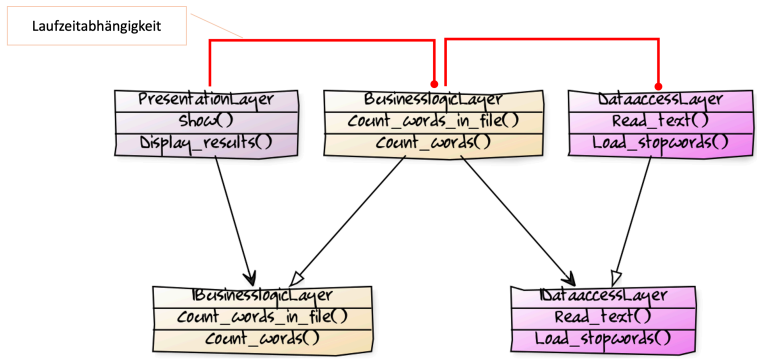


Abbildung 12: Mit DIP unterscheiden sich die Abhängigkeiten zu Compilezeit und Laufzeit

Das sieht jetzt schon nicht mehr so einfach aus wie das ursprüngliche Schichtenmodell, würde ich sagen. Logik ist auch im Schichtenmodell bei vortrefflicher Ausrichtung der Beziehungen immer noch funktional abhängig. Um trotz dieser Abhängigkeiten Testbarkeit zu erlangen, müssen zusätzliche Elemente eingeführt werden: Abstraktionen (hier: Interfaces). Und aus einer Menge unidirektionaler Abhängigkeiten werden zwei Mengen, von denen eine auch noch gegenläufige Abhängigkeiten enthält.

Das scheint der Preis der Wandelbarkeit zu sein. Verständlichkeit entsteht durch Trennung von Verantwortlichkeiten und klare Beziehungen. Testbarkeit entsteht durch DIP und IoC. Ist halt so. Da müssen wir durch.

Um das Leben nun wenigstens aber ein wenig einfacher zu machen, gibt es Mock-Frameworks wie Moq[11] (in Abbildung 10 benutzt) und Dependency-Injection-Frameworks wie Simple Injector[12] oder Unity[13].

Jetzt ist es nur noch eine Sache konsequenter Anwendung von Prinzipien und Werkzeugen, um sauberen Code zu schreiben. Alles scheint gesagt zur grundlegenden Strukturierung von Logik.

Das ist zumindest der Stand des Bewusstseins, den ich bei Clean Code Development Trainings[14]. Wenn ein Architekturmuster bekannt ist, dann ist es MVC oder das Schichtenmodell. Zusätzlich wird dann noch die Fahne der SOLID-Prinzipien[15] hochgehalten, zu denen das DIP gehört wie auch das *SRP (Single Responsibility Principle)*, das sich mit Verantwortlichkeitstrennung befasst.

Nur leider sehe ich ebenfalls bei den Teams, die Code SOLIDe in Schichten strukturieren keine entspannten und freudvollen Gesichter. Der Code ist immer noch schwer zu wandeln. Sonst würde man mit mir ja auch nicht über Clean Code Development sprechen wollen.

Wie kann das aber sein? Trotz sauberer Schichtung immer noch nicht sauber? Merkwürdig, oder?

Schichten in Schale werfen

[11] https://github.com/Moq/moq4
[12] https://simpleinjector.org/index.html
[13] https://github.com/unitycontainer/unity
[14] https://ralfw.de/trainings/training-products/
[15] https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Reflexion

Früher war nicht alles besser. Codezustände wie in Abbildung 1 will niemand (wieder) haben.

Die erste Variante des Schichtenmodells jedoch, die war gar nicht so schlecht. Die klare Trennung von Verantwortlichkeiten kombiniert mit einer konsequenten Ausrichtung der Abhängigkeiten hat die Verständlichkeit deutlich gesteigert (Abbildungen 4 und 5).

Für gute Wandelbarkeit war das allerdings noch nicht genug. Die braucht nicht nur Verständlichkeit, sondern auch Testbarkeit. Die war in der ersten Schichtenvariante noch begrenzt. Lediglich die Logik der untersten Schicht war gut testbar, weil sie für sich stand. Die Logik in den darüber liegenden Schichten konnte zwar grundsätzlich schon gezielt angesprochen werden, nur musste dann auch immer die Logik darunter liegender Schichten beim Test mit durchlaufen werden. Das kostet Zeit und das macht es nicht leicht, einen Bug zu lokalisieren.

Die Testbarkeit ist dann in der zweiten Variante des Schichtenmodells (Abbildungen 8 und 9) nachgezogen worden. Durch Anwendung von DIP und IoC können für Tests untere Schichten ausgeblendet werden. Wenn etwas schiefgeht, dann weiß man, dass der Fehler in der Logik der zu testenden Schicht steckt.

Allerdings: Dieser Fortschritt in der Testbarkeit hat seinen Preis. Der wird deutlich in der Clean Architecture Variante (Abbildungen 15 und 19). DIP und IoC addieren Komplexität, die die Verständlichkeit nun – zumindest nach meinem Empfinden – massiv reduziert. Man ist über das Ziel hinausgeschossen. „Mehr vom Selben“ (hier: DIP und IoC) hat den Fortschritt, den die zweite Variante des Schichtenmodells gebracht hat, nicht vergrößert. Im Gegenteil!

Aber wie kommt das? Ich glaube, es liegt daran, dass man zu sehr auf die Compilezeitabhängigkeiten gestarrt hat.

Von der Schichtung zur Konzentrik überzugehen hat etwas mit dem Compilezeitabhängigkeiten zu tun. Im Schichtenmodell war das Volatile (oder Instabile), Robert C. Martins „mechanisms“, nicht konsequent in einer Position, wo Veränderungen wenig Probleme machen.

Relativ problemlos sind Veränderungen nämlich dort, wovon nur wenige oder keine Codeeinheiten abhängig sind. Im Schichtenmodell ist das nur für die Präsentationslogik der Fall. Die Datenzugriffslogik hingegen, die ebenfalls ein „mechanism" ist, muss wegen der Abhängigkeit anderer von ihr, Stabilität zusichern.

Das wurde mit der Clean Architecture bewusst verändert. Dort sind nicht nur die Abhängigkeiten sauber ausgerichtet, sondern auch die Verantwortlichkeiten nach Stabilität positioniert: am stabilsten sind ganz allgemeine, grundlegende Regeln im Kern, am instabilsten die Kommunikation mit der Umwelt in der äußeren Schale.

Diese Sichtweise gefällt mir – allerdings hat die Implementation eben einen hohen Preis. Abbildungen 17 und 20 machen es exemplarisch deutlich: die Verständlichkeit der Zusammenhänge im Code sinkt.

Aber selbst wenn ich einmal über den auch von Robert C. Martin beklagten Mehraufwand hinwegsehe, frage ich mich, was wirklich gewonnen ist. Denn Abbildung 21 zeigt ein Bild, das sich im Grunde nicht vom Schichtenmodell unterscheidet. Zur Laufzeit ist eine Businesslogik immer noch vom Datenzugriff abhängig.

Was soll das? Da mag zur Compilezeit die Abhängigkeit umgekehrt sein – Adapter außen hängt von Domäne innen ab –, doch zur Laufzeit gehen die Aufrufe dorthin durch die Domäne in die Tiefe.

Es ist letztlich nichts gewonnen. Die Abstraktion, von der die Businesslogik abhängig ist, ist lediglich verschoben worden. Vorher gehörte sie zur darunterliegenden Schicht (*IDataaccessLayer* in Abbildung 9), jetzt gehört sie zur Businesslogik selbst (siehe *IStopwords* in Abbildung 19).

Dass nichts gewonnen ist, ist deutlich zu bemerken beim Testen. Das Architekturbild suggeriert, dass eine innere Schale keine Abhängigkeit hat zu einer äußeren – doch beim Testen stellt sich das Gegenteil heraus. In Abbildung 17 ist der *Use Case Interactor* – also Code der Use-Case-Schale – zur Laufzeit abhängig vom *Presenter* in der darüberliegenden Schale. Dass der *Use Case Output Port* zur Use-Case-Schale gehört, kaschiert das nur. Im Test muss trotzdem eine Attrappe gebaut werden.

Sind Sie noch da oder haben Sie schon halb abgeschaltet? Das würde mich nicht wundern. Bei dem ganzen hin und her der Abhängigkeiten von Compilezeit und Laufzeit, kann einem schon der Kopf schwirren. Ich

jedenfalls verliere bei der Darstellung der vollständigen Abhängigkeiten der implementierten Clean Architecture den Überblick (Abbildung 22).

So sieht für mich ein Testalbtraum aus. Ganz zu schweigen davon, dass damit das Rätselraten darum, für welche Klassen ein Interface definiert werden sollte, weiter angeheizt wird. Im Zweifelsfall lautet dann die Antwort „Für alle!“ und damit explodiert die Zahl der Dateien in einem Projekt (wenn man je Klasse und je Interface eine Datei denkt).

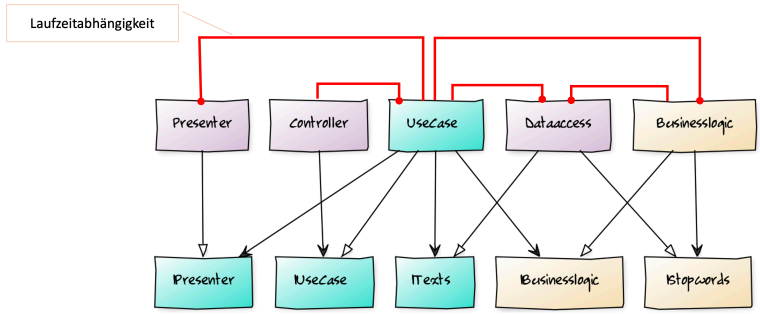


Abbildung 22: Das vollständige Abhängigkeitsbild der implementierten Clean Architecture

Das einzig Positive, das ich der Clean Architecture abgesehen vom hübschen Bild abgewinnen kann, ist die Tendenz zur Aufspaltung von Abstraktionen. Sie scheint die Anwendung des *Interface Seggregation Principle (ISP)*[16] nahezulegen. Schmalere Interfaces helfen einfach bei der Entkopplung.

Wo vorher nur eine Präsentationslogikschicht und eine Datenzugriffsschicht mit respektiven Abstraktionen waren, sind beide nun zerfallen in mehrere Teile: *Controller* und *Presenter* sind getrennt und es gibt *IPresenter*; ebenfalls getrennt sind *IText* und *IStopwords*, wo vorher nur *IDataAccessLayer* war.

Am Ende ist dieser positive Effekt für mich jedoch nicht ausschlaggebend. Viel wichtiger finde ich das Sinken der Verständlichkeit durch eine gestiegene Artefaktzahl und den weiterhin hohen Testaufwand. Denn jede Laufzeitabhängigkeit ruft zur Testzeit nach einer Attrappe.

[16] https://en.wikipedia.org/wiki/Interface_segregation_principle

2 - Das IODA Architekturmodell

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Funktionale Abhängigkeiten als Wurzelproblem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Auflösung funktionaler Abhängigkeiten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Operationen verbinden

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Verbindung zur Außenwelt

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Ein neues Architekturmuster

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Echt abstrakt

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

3 - IODA am Beispiel

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Struktur fraktal

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Zusammenfassung

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Update 2020

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Logik frisch definiert

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Integrationen konsequent benannt

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Interactor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Processor

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Interactor-Varianten

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.

Reflexion

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/ioda-architektur-im-vergleich-dnp.