LANGLEBIGE SOFTWARESTRUKTUREN SCHAFFEN, TEIL 1

Architektur blitzblank

In puncto Softwarearchitekturen gilt es viele Dinge zu beachten. Deren Zusammenhänge, chronologische Entwicklung und Auswirkungen klärt diese zweiteilige Artikelserie.

Fred Brooks erläuterte in seinem Buch „The Mythical Man-Month“ [1], warum Softwareprojekte fehlschlagen. Dabei kam er zu dem Schluss, dass in den seltensten Fällen der Grund für das Scheitern technischer Natur ist, die inhärente Komplexität der betroffenen Systeme dann aber wiederum hauptausschlaggebend ist. Dies ergibt sich daraus, dass Entwickler ständig die komplexen Zusammenhänge entwirren müssen, wenn sie den vorliegenden Code lesen, um ihn verstehen zu können, und sie müssen den Code verstehen, um ihn überhaupt erst einmal verändern zu können. Demzufolge ist komplex zu verstehender Code auch komplex zu ändern, und alles, was komplex ist, birgt eben die Gefahr von Fehlern. Es ergibt sich somit eine Art von Komplexitätsspirale, der man so früh wie möglich entgegenwirken muss.

Diese Artikelserie geht daher auf einen Streifzug durch die entsprechenden Gegenmaßnahmen, die wir in der Softwareentwicklung erarbeitet haben, hebt deren Auswirkungen auf die Softwarearchitektur hervor und spannt damit einen Bogen über die Arbeiten von bekannten Autoren wie Robert C. Martin, Martin Fowler und Eric Evans. Hierzu geht dieser erste Teil zunächst auf die typischen Grundlagen wie Entwurfsmuster und die weit verbreitete Schichtenarchitektur ein. Ausgehend von deren Nachteilen startet der zweite Teil dann in fortgeschrittenere Architekturmuster, bis hin zur Clean Architecture.

Mustererkennung als Grundlage

Einmal ganz direkt gefragt: Waren die Eingangssätze zur Komplexität für Sie leicht zu verstehen? Wenn nicht, ist dies nur natürlich und erlaubt uns, zusammen mit diesem Absatz etwas zu beobachten, das wir im Quellcode häufiger vorfinden: sich selbst referenzierende Strukturen. Oft ist es so, dass wir Methoden lesen, die andere Methoden aufrufen. Wir springen von einem Bereich des Textes (oder eben Quellcodes) zu einem anderen und wieder zurück. Dabei tun wir etwas, was unser Gehirn ganz und gar nicht mag. Einem Compiler gleich müssen wir uns Absprungmarken und Sprungziele merken, die wir im Kopf wie in einem Stack organisieren. Leider ist unser Gehirn rein evolutionär nicht gut darin, diesen Stack abzuarbeiten und sich alle Sprungmarken zu merken. Ab einer gewissen Größe tun wir uns schwer mit langen Kausalketten voller logischer Rücksprünge und Verzweigungen. Nachzuvollziehen, was im Code mehrere Methoden, Klassen, Module, Komponenten oder Applikationen tun, bedeutet Stress für unser Gehirn, und daher brauchen wir alle Hilfe, die wir bekommen können, um diese Belastung zu meistern. Aus diesem Grund ist unsere Industrie auch so bemüht, Code zu vereinfachen. Damit ist nicht nur die Industrie gemeint, die versucht Low-Code-Lösungen zu etablieren oder Entwicklern mit Entwicklungsumgebungen unter die Arme zu greifen. Nein, gemeint damit ist die Gesamtheit aller Entwickler, die über Programmierparadigmen und Patterns den Code direkt selbst vereinfacht. Hierbei greift in aller Regel etwas, in dem unser Gehirn wiederum wirklich gut ist: die Mustererkennung. Das war schon hilfreich, als wir unser Abendessen nicht indirekt mit der Tastatur, sondern ganz direkt mit dem Bogen fingen oder von einem Strauch zupften. Während Muster uns damals dabei halfen, die guten Beeren von den schlechten zu unterscheiden, helfen sie uns heute dabei, Quellcode in Kategorien zu unterteilen. Mehr noch, sie helfen uns dabei, diese Kategorien zu definieren und gezielt nach ihnen zu entwickeln. Letztendlich ist die objektorientierte Programmierung selbst ein Ergebnis der Bestrebung,

Klassen, Komponenten, Module und Pakete?

Leider sind wir in der Softwareentwicklung recht nachlässig, was das Thema Definitionen angeht, und daher werden Begriffe gern eher lax verwendet, statt sie genau zu beschreiben. So auch bei der Frage, wo eigentlich der Unterschied zwischen Klassen, Komponenten, Modulen und Paketen liegt. Klassen müssen wir hierbei wahrscheinlich nicht ausgiebiger erklären. Komponenten hingegen sind Cluster von Klassen, die dem gleichen Ziel dienen. Ein Softwaremodul ist im Grunde das Gleiche, aber ohne dieses gebundene Ziel. Je nachdem, auf welcher Abstraktionsebene man sich befindet, kann es daher sein, dass Module Komponenten zusammenfassen oder Komponenten sich aus mehreren Modulen bedienen. Noch verrückter wird dieser Wirrwarr, wenn man bedenkt, dass Microsoft jede einzelne DLL als eigenes Softwaremodul bezeichnet. Dies ist deshalb so verrückt, da es dafür eigentlich den Begriff des Pakets gibt. Ein Paket ist ein auslieferbares und physisch auf der Festplatte vorhandenes Softwarebündel. Darin können mehrere Module und/oder Komponenten, ja sogar DLLs zusammengefasst werden.

In diesem Artikel werden die Begriffe Module und Komponenten meist synonym verwendet und es wird davon ausgegangen, dass Komponenten eine feinere Granularität und stärkere innere Bindung haben als Module.

gleichartige Logikblöcke auch logisch gemeinsam zu verwalten. Es gibt aber auch noch andere Ausprägungen. Wir regeln zum Beispiel mit Coding Guidelines die Benennung, die Abstände und die Reihenfolge von Klassenbestandteilen. Damit bringen wir den Code rein visuell schon in ein bestimmtes Muster. Darüber hinaus halten wir durch sie auch unsere Methoden kurz und unsere Abstraktionsebenen getrennt, was der Verständlichkeit dient. Des Weiteren kennen wir Code Smells, die nichts anderes sind als Muster, die auf mögliche Probleme hinweisen, und Refactorings, die uns helfen, den Code von den ungewollten in die gewollten Muster zu übertragen. Martin Fowler hat hierzu ein großartiges Buch geschrieben, das beide Kategorien einander gegenüberstellt [2]. Dank diesem einfachen Handwerkszeug können wir Entwickler also Code schreiben, der auf der Ebene der Methoden und Klassen leicht zu verstehen ist. Das ist richtig, wichtig und auch gut so. Nur ist es leider nicht ausreichend.

Entwurfsmuster

Die schmerzhafte Erkenntnis, dass Code, der den Coding Guidelines folgt, nicht automatisch auch verständlicher Code ist, trifft die meisten Entwickler eher verspätet. Dies kommt daher, dass man viele Monate nach der Programmierung nicht mehr über alle Implementierungsdetails verfügt. Man muss sich dann oft das Bild des großen Ganzen erst einmal wieder erarbeiten. Dabei helfen sprechende Namen von Methoden und Variablen zwar, wenn diese Namen aber zu wenige Hinweise auf den Gesamtkontext geben, bieten sie einen geringen Mehrwert. Hinzu kommt, dass sich dieser Gesamtkontext meist erst aus den Interaktionen aller Codebestandteile ergibt und Rückschlüsse auf die realen Anforderungen geben muss. Damit ergibt sich ein Großteil der Komplexität von Quellcode nicht anhand des Aufbaus einzelner Codebestandteile, sondern aus deren Abhängigkeiten zueinander und Interaktionen untereinander.

Diese Interaktion von Codebestandteilen ist einer der wichtigsten Faktoren, die den Paradigmenwechsel von den Tagen der imperativen Programmierung über objektorientierte Techniken bis hin zu funktionalen Varianten vorangetrieben haben. Von vergleichsweise simplen Ansätzen hat sich unsere Branche zu bekannten Entwurfsmustern entwickelt, die ein Problem beschreiben und eine wiederverwendbare Form der Lösung bieten. Entwurfsmuster weisen dabei eine höhere Abstraktionsebene auf als der reine Quellcode und können verwendet werden, um die Interaktionen einer Handvoll von Klassen zu beschreiben. Diese Muster können somit als eine Art standardisierter Bausteine verstanden werden, um Lösungen zu erstellen, die leichter zu verstehen sind. Hierbei hilft, dass die dahinterliegenden Konzepte ausführlich in Standardwerken wie den „Design Patterns“ der Gang of Four (GoF) [3] oder den „Patterns of Enterprise Application Architecture“ (PoEAA) von Martin Fowler [4] beschrieben sind.

Softwarearchitektur

Moderne Systeme können jedoch leicht Hunderttausende oder sogar Millionen von Codezeilen in Tausenden von Klassen enthalten. Selbst wenn diese Klassen mithilfe von Entwurfsmustern in Hunderte von Komponenten aufgeteilt werden, ist das immer noch zu viel, um ein konsistentes Gesamtbild zu vermitteln. An dieser Stelle kommt die Softwarearchitektur ins Spiel. Sie befasst sich mit dem großen Ganzen. Man könnte auch sagen, dass sie sich auf einer Metaebene über den Entwurfsmustern bewegt. Sie mappt dabei die High-Level-Struktur des Softwaresystems mit den funktionalen und nicht funktionalen Anforderungen. Zu diesen gehören beispielsweise Zuverlässigkeit, Sicherheit, Wartbarkeit und Erweiterbarkeit. Sie befasst sich mit Entscheidungen, die in Bezug auf diese Aspekte kritisch sind und sich während der Entwicklung nur schwer ändern lassen, wie zum Beispiel die Wahl der Programmiersprache oder der Bereitstellungsmethode.

Konzepte für sauberen Code sind auf Ebene der Architektur leicht zu erkennen: Namenskonventionen und Ansätze zur Codeorganisation sind weit verbreitete Anliegen, die häufig als „architecturally evident coding“ (architektonisch offensichtliche Codierung) bezeichnet werden. Auch das Konzept der Muster taucht auf dieser höheren Abstraktionsebene in Form von Architekturmustern wieder auf. Zu diesen zählen beispielsweise Model-View-Controller oder Model-View-ViewModel. Softwarearchitekten müssen jedoch weit mehr als diese Faktoren berücksichtigen.

Agile Softwareentwicklung, Continuous Delivery und die DevOps-Bewegung haben auch den Schwerpunkt der Architektur verschoben. Die alte Praxis des großen Entwurfs im Voraus verträgt sich nicht gut mit einigen agilen Konzepten ▶

Die SOLID-Prinzipien

Die SOLID-Prinzipien wurden von Robert C. Martin zusammengefasst [7] und adressieren die Grundprinzipien, nach denen Klassen, Module beziehungsweise Komponenten designt sein sollten. Die nachfolgenden Definitionen sind nicht eindeutig, da sich die Beschreibungen über die Zeit teils geändert haben:

- **Single Responsibility Principle (SRP)** – „Eine Klasse sollte einen – und nur einen – Grund haben, sich zu ändern.“ Später geändert in: „Ein Modul sollte einem – und nur einem – Akteur gegenüber verantwortlich sein.“
- **Open Closed Principle (OCP)** – „Das Verhalten einer Klasse sollte sich erweitern lassen, ohne sie zu modifizieren.“
- **Liskov Substitution Principle (LSP)** – „Abgeleitete Klassen müssen durch ihre Basisklassen ersetzbar sein.“
- **Interface Segregation Principle (ISP)** – „Erstelle feingranulare Schnittstellen spezifisch für ihre Nutzer.“ Gegebenenfalls auch: „Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden.“
- **Dependency Inversion Principle (DIP)** – „Man soll von Abstraktionen abhängen, nicht von Details“ oder detaillierter: „Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen. Abstraktionen sollten nicht von Implementierungsdetails abhängen. Implementierungsdetails sollten von Abstraktionen abhängen.“

wie häufige Releases und kurze Planungshorizonte. Die agile Entwicklung drängt die Teams dazu, endgültige Entscheidungen bis zum spätestmöglichen sinnvollen Moment aufzuschieben und damit die Zeit zu maximieren, in der alle für die Entscheidung relevanten Informationen gesammelt werden können [5]. Agiles Vorgehen zwingt die Architektur also dazu, „elastisch" und damit offen für Veränderungen zu sein. Damit folgt die Softwarearchitektur dem gleichen Pfad wie der Rest der Softwareentwicklung und wird zu einem inkrementellen Vorgang. Inkrementelle Architektur erfordert ständige Kommunikation zwischen denjenigen, die die Interna der Software kennen und wissen, wie man Software schreibt, und denjenigen, die wissen, was die Software tun soll und welchen externen Einflüssen sie sich ausgesetzt sieht. Um diese Interaktion so einfach wie möglich zu gestalten, muss die dabei automatisch entstehende Sprachbarriere zwischen beiden Welten überbrückt werden. Eine Aufgabe, die mindestens, aber nicht ausschließlich den Architekten und Testern zufällt.

Entwickler müssen ihr Wissen über die Domäne und das Unternehmen auf den Quellcode abbilden. Daher müssen sie sich in der Anwendung so gut zurechtfinden, dass sie das Code-Äquivalent der Geschäftskonzepte und -regeln finden können. Es ist die Aufgabe der Architektur, den Entwicklern diese Sicherheit zu geben. Nach Brooks stellt die Architektur eines Softwaresystems eine Gesamtvision dessen dar, was das System tun soll und wie es dies tun soll. Die Architektur sollte die Anwendungsfälle, für die das System geschaffen wurde, klar kommunizieren. Mit anderen Worten: Zwei Softwaresysteme in derselben Domäne (zum Beispiel Halbleiterfertigung), die mit zwei deutlich unterschiedlichen Technologiestacks (zum Beispiel Python und ASP.NET) erstellt wurden, sollten hinsichtlich ihrer Architektur viel mehr Gemeinsamkeiten aufweisen als zwei Anwendungen in zwei verschiedenen Domänen (zum Beispiel Halbleiterfertigung und Augenheilkunde), die dieselbe Technologie (zum Beispiel ASP.NET) verwenden. Brooks nennt dieses Merkmal „konzeptionelle Integrität", während Robert C. Martin eher den Begriff „schreiende Architektur" (Screaming Architecture) [6] verwendet, um diesen Aspekt zu beschreiben.

Die Paketierungsprinzipien

Diese Prinzipien sind entkoppelt von den bekannten SOLID-Prinzipien und beschreiben vor allem die Kohäsion innerhalb von auslieferbaren Paketen [7]:

- **Release/Reuse Equivalency Principle (REP)** – „Die Granularität der Wiederverwendung ist die Granularität des Releases."
- **Common Closure Principle (CCP)** – „Klassen, die sich gemeinsam verändern, sollten gemeinsam paketiert werden."
- **Common Reuse Principle (CRP)** – „Die Klassen in einem Modul werden gemeinsam wiederverwendet. Wenn eine Klasse eines Moduls wiederverwendet wird, werden alle wiederverwendet."

Alles in allem hat die Softwarearchitektur eine ganze Reihe von Aufgaben zu erfüllen:

- die Komplexität zähmen,
- die Absicht und die Anwendungsfälle des Systems klar kommunizieren,
- Konventionen und Code-Organisationsstrukturen für eine einfache Navigation innerhalb des Codes bereitstellen,
- die Flexibilität bieten, Entscheidungen zu verschieben, um ein inkrementelles Design zu ermöglichen.

Traditionelle Praktiken und Muster, wie die klassische Schichtenarchitektur, können Schwierigkeiten haben, all diese Anforderungen zu erfüllen.

Die SOLID-Prinzipien

Wir haben bereits gesehen, dass sich sauberer Code auch auf die Softwarearchitektur erstreckt, da bestimmte Codierungskonzepte wie Namenskonventionen und Codestruktur-Anordnungen von der Ebene der Methoden und Klassen auf die Ebene der Komponenten, Module und Anwendungen projiziert werden. Ein Begriff, über den man in diesem Zusammenhang häufiger stolpern wird, sind die SOLID-Prinzipien. Bei ihnen handelt es sich um Leitlinien für das Schreiben von besserem Code, die von Robert C. Martin zusammengetragen wurden. Auch wenn nicht ganz sicher ist, inwieweit SOLID sich auf eine komplette Softwarearchitektur anwenden lässt, können wir daraus doch zumindest einige Regeln ableiten, die erheblichen Einfluss auf die entstehenden Softwarestrukturen haben. Da wir nachfolgend mehrfach auf die SOLID-Familie verweisen, werden diese Prinzipien im Kasten Die SOLID-Prinzipien einmal dargestellt und nachfolgend zusammengefasst.

Das Single-Responsibility-Prinzip (SRP) adressiert den inneren Zusammenhalt aller Bestandteile, der auch als Kohäsion bezeichnet wird. Bei einer geringen Kohäsion gibt es Teile, die nichts oder nur sehr wenig miteinander zu tun haben und daher unterschiedliche Gründe für Änderungen haben können. Eine hohe Kohäsion bedeutet wiederum, dass sich alle Teile auf die Erfüllung der gleichen Aufgabe konzentrieren. Daher ist die Summe aller Teile leichter zu verstehen, anzupassen und gegebenenfalls wiederzuverwenden. Das Open/Closed-Prinzip (OCP) bedeutet, dass das Verhalten erweitert werden kann, ohne es in seinem Grundaufbau zu verändern. Dies ist deshalb so wichtig, da eine Veränderung des Verhaltens beziehungsweise des Quellcodes immer auch eine Änderung der vorhandenen Konsumenten nach sich zieht. Eine Erweiterung beeinflusst die vorhandenen Konsumenten nicht, da sie jene Erweiterung einfach nicht verwenden. Das Liskov Substitution Principle (LSP) verlangt, dass alle erweiterten Typen ohne Änderung auch dort eingesetzt werden können, wo ihre Basistypen eingesetzt werden. Das Interface Segregation Principle (ISP) wiederum ermutigt Entwickler, mehrere kleine, kohärente Schnittstellen zu erstellen und sie auf der Grundlage der Anforderungen des nutzenden und nicht des bereitstellenden Programmteils zu definieren. Das Dependency Inversion Principle (DIP) führt die anderen Prinzipien zusammen, indem es fordert, dass die Module oder

Die Kopplungsprinzipien

Dies sind ebenfalls Prinzipien, die von Robert C. Martin formuliert wurden [7] und in diesem Fall die Abhängigkeiten zwischen Bestandteilen näher beschreiben:

- **Acyclic Dependencies Principle (ADP)** – „Abhängigkeitsgraphen von Paketen dürfen keine Kreise aufweisen."
- **Stable Dependencies Principle (SDP)** – „Abhängigkeiten sollten in der Richtung der Stabilität verlaufen."
- **Stable Abstractions Principle (SAP)** – „Die Abstraktheit nimmt mit der Stabilität zu."

Klassen der höheren Ebene Abstraktionen definieren (wie im ISP deklariert) und von diesen Abstraktionen abhängen anstatt von den Implementierungen dieser Abstraktionen (wie im OCP deklariert).

SOLID weitergedacht

Beim Lesen der SOLID-Prinzipien wird klar, dass sie sich stark auf die Struktur eines Softwaresystems auswirken und damit direkt die Architektur beeinflussen. Immerhin arbeiten Klassen, die unter Berücksichtigung von SOLID entworfen wurden, ganz anders zusammen als Klassen, die nicht nach diesen Grundsätzen erstellt wurden. Wenn wir jedoch explizit über Komponenten anstelle von Klassen nachdenken, haben diese Grundsätze einige zusätzliche Konsequenzen.

Das Single Responsibility Principle kann leicht von Klassen auf Komponenten übertragen werden: Komponenten sollten kohärent sein. Wenn wir jedoch auch Open/Closed in die Betrachtung mit aufnehmen, kommen wir zu dem Schluss, dass physisch oder konzeptionell miteinander verbundene Klassen sich wahrscheinlich auch gemeinsam ändern werden. Diese Beobachtungen kommen im Common Closure Principle (CCP) zum Ausdruck (siehe Kasten Die Paketierungsprinzipien), das uns dazu veranlasst, Komponenten zu erstellen, die stark miteinander verbundene Klassen zusammenfassen und somit selbst eine hohe Kohäsion aufweisen.

In ähnlicher Weise wirkt sich das Interface Segregation Principle auf Komponenten aus und führt uns zum sogenannten Common Reuse Principle (CRP). Beide scheinen zwei Seiten derselben Medaille darzustellen und haben im Hinblick auf transitive Abhängigkeiten und das Deployment von Paketen ganz klare Auswirkungen auf verschiedene Aspekte der Softwarearchitektur. Hängt ein Paket, ein Modul, eine Komponente, eine Klasse oder eine Schnittstelle von einem Bestandteil ab, so hängt es auch transitiv von allen Elementen ab, die Abhängigkeiten des Bestandteils darstellen.

Gerade bei externen Bibliotheken wie NuGet-Paketen wird das deutlich. Trennt man beispielsweise Schnittstellen nicht von ihren Implementierungsdetails und gestaltet sie zu groß, ziehen sich diese Datentypen bis in das Deployment durch. Dann kann es passieren, dass man für einfache Unit-Tests externe Bibliotheken einbinden muss, nur weil man etwas prüfen möchte, dessen Schnittstellen Datentypen dieser Bibliotheken verwenden. Insbesondere bei OR-Mappern wie Entity Framework oder bei SDKs für zum Beispiel Azure steht man dann sehr schnell vor dem Problem, dass man den Code nicht testen kann, da die verwendeten externen Bibliotheken eine Umgebung erwarten, welche man im Test nicht nachstellen kann. Hieran zeigt sich dann auch eine Verletzung des Dependency Inversion Principle, das ja genau vorgibt, dass Schnittstellen nicht von solchen Details abhängen sollen.

Übertragen auf die Welt der Komponenten und Module heißt das: „Module, die Implementierungsdetails enthalten, sollten von Modulen abhängen, die Abstraktionen enthalten, und nicht umgekehrt", oder kürzer ausgedrückt: „Module sollten in Richtung der Abstraktion abhängen." Dieses Prinzip beschreibt Martin als Stable Dependencies Principle (SDP), welches zur Gruppe der Kopplungsprinzipien gehört (siehe Kasten Die Kopplungsprinzipien).

Was kann man nun aus diesen Betrachtungen lernen? Zum einen, dass sich mit der Perspektive auch die Gesetzmäßigkeiten und Prinzipien schrittweise weiterentwickeln. Sie sind zwar noch gleich, aber erhalten immer weitere Details zu Aspekten, die auf Ebene der Klassen und Methoden noch nicht ins Gewicht fallen. Dazu gehören eben die Paketierung und das Deployment, aber auch die veränderte Art der Wiederverwendung und die sich daraus ergebenden Abhängigkeiten. Eine Klasse ist nun einmal keine Komponente, sondern in aller Regel ein Teil einer Komponente.

Besonders deutlich wird dies beim Thema Abstraktion. Während eine Klasse entweder abstrakt oder konkret ist und es kein Dazwischen gibt, ist diese Unterscheidung bei Komponenten, die mehrere Klassen und Schnittstellen enthalten, selten binär: Die Abstraktheit einer Komponente kann auf einer Skala zwischen den Extremen „maximal abstrakt" und „maximal konkret" gemessen werden, je nachdem, wie hoch der Anteil der konkreten beziehungsweise abstrakten Typen ist. Auf eine ähnliche Weise lassen sich dann auch die Abhängigkeiten betrachten, die entweder darin bestehen, dass die Komponente etwas verwendet oder verwendet wird. Martin leitet daraus einen Wert für die Stabilität der Komponente ab, indem er sagt, dass Elemente mit vielen Abhängigkeiten besonders instabil sind, da jede Abhängigkeit ihnen Anlass zu einer Änderung geben kann. Elemente, die von nichts abhängen, sind wiederum stabil, da nur ihre eigenen Belange einen Anlass für eine Änderung bieten. All diese Zusammenhänge stellt Martin mit der Abstraktion ins Verhältnis, um ableiten zu können, ob die Module eines Systems adäquat geschnitten sind (siehe Bild 1). In diesem Fall liegen sie auf der sogenannten Main Sequence.

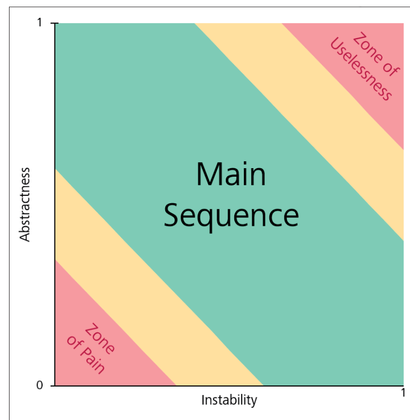
Aufgrund der Komplexität der Berechnung sparen wir aber das konkrete Vorgehen aus und begnügen uns mit folgender Erweiterung des Dependency Inversion Principle: Abstraktionen sollten möglichst wenig Abhängigkeiten haben und konkrete Implementierungen möglichst von Abstraktionen abhängen. Das führt uns zum Stable Dependencies Principle (SDP), wonach Abhängigkeiten immer entlang ihrer Stabilität verlaufen. Oder ganz kurz zusammengefasst: Module sollten nur einen Grund haben, sich zu ändern, sie sollten keine Abhängigkeiten von Komponenten schaffen, die sie nicht ▶

brauchen, sie sollten in Richtung der Abstraktion und Stabilität abhängen und dabei ebenso stabil wie abstrakt sein.

Warum genau haben wir nun aber all das betrachtet? Zum einen, um zu zeigen, dass es neben den SOLID-Prinzipien noch weitere Prinzipien gibt, die auf ihrer jeweilen Abstraktionsebene von Bedeutung sind. Zum anderen aber auch, um Kriterien zu schaffen, mit denen wir verbreitete Architekturmuster bewerten können. In diese Bewertung fließen darüber hinaus noch die generellen Anforderungen an eine Softwarearchitektur ein. So haben wir uns eine Architektur zum Ziel gesetzt, die die Komplexität eindämmt, die Absicht und die Anwendungsfälle des Systems klar kommuniziert, die Konventionen und Code-Organisationsstrukturen für eine einfache Navigation bereitstellt und die Flexibilität bietet, Entscheidungen zu verschieben.

Die Schichtenarchitektur

Bevor sich die objektorientierte Programmierung durchsetzte, waren die meisten Anwendungen imperativ oder prozedural und führten Stapeloperationen aus. Daten wurden aus Dateien gelesen, geändert und in Dateien geschrieben. Für diese Operationen war nicht viel Strukturierung notwendig. Später, als sich in den 90er-Jahren die Client-Server-Architektur durchsetzte, musste sich die Anwendungsstruktur erheblich ändern. Die Daten befanden sich auf den Servern, die Klienten stellten die Benutzeroberfläche dar, und in der Regel handelte es sich bei den beiden um zwei physisch getrennte Computer. Die Geschäftslogik wurde entweder direkt in der Serverkomponente, die ansonsten für die Persistenz zuständig war, oder in der Clientkomponente, die ansonsten für die Präsentation zuständig war, programmiert. Die Entscheidung, wo genau die Geschäftslogik zu platzieren war, beruhte in aller Regel auf Faktoren wie Gesamtkomplexität der Lösung, Leistung der ausführenden Laufzeitumgebung und eventuell Sicherheitsregularien. Je mehr Geschäftsprozesse umgesetzt wurden und je komplexer diese waren, desto wertvoller wurde die Geschäftslogik und desto anfälliger wurde sie für Änderungen. Das machte sie auf Dauer instabil und schwer zu verstehen.



Abstraktheit versus Instabilität (Bild 1)

Die Lösung für diese Probleme brachte die objektorientierte Programmierung. Bei der objektorientierten Programmierung war es nicht mehr erforderlich, den Code in der Reihenfolge seiner Ausführung anzuordnen, sodass die Geschäftslogik von den anderen Aufgaben des Quellcodes getrennt werden konnte. Dieses Muster wurde als Drei-Schichten-Architektur bekannt und ist selbst in modernen Softwaresystemen immer noch häufig anzutreffen. Je nach Einsatzgebiet ist dies weder verwerflich noch falsch, in sehr großen Systemen mit vielen Entwicklern kann es aber zu erheblichen Problemen führen.

Schichtenarchitekturen definieren Strukturen auf hoher Ebene, die als Schichten oder Layer bezeichnet werden und jeweils auf einen bestimmten Aspekt des Programms spezialisiert sind, zum Beispiel Präsentation, Geschäftslogik oder Persistenz. Jede Schicht abstrahiert von der Arbeit, die zur Erfüllung einer bestimmten Aufgabe erforderlich ist, und überlässt die Details den darunterliegenden Schichten. Damit ist eine Schicht also eine Ebene, in der alle Elemente den gleichen Abstraktionsgrad besitzen. So muss die Geschäftslogikschicht beispielsweise nur wissen, dass sie Daten bei der Persistenzschicht abfragen soll, und sie muss nicht wissen, woher diese Daten kommen oder wie sie zu beschaffen sind. Sie führt lediglich die erforderlichen Geschäftsvorgänge aus und gibt die verarbeiteten Daten an die Präsentationsschicht weiter, die sich um die Formatierung für den Benutzer kümmert. Aufgrund dieser Trennung von Belangen ermöglicht die Schichtenarchitektur ein kohärenteres Design für jeden Aspekt, wodurch die Entwürfe leichter zu interpretieren sind. Außerdem wird die Architektur dadurch konformer mit dem Common Closure Principle: Jede Schicht hat nur wenige Gründe, sich zu ändern.

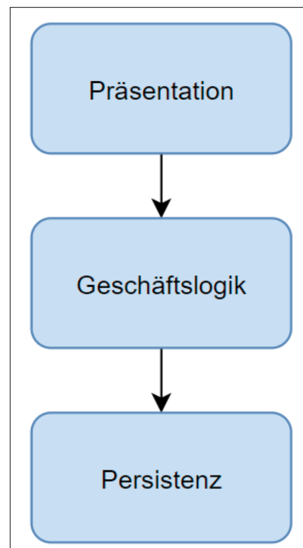
Schichten werden übereinandergestapelt dargestellt (siehe Bild 2). Die Abhängigkeiten zwischen ihnen dürfen nur nach unten zeigen, sodass eine Komponente in einer Schicht nur auf andere Komponenten in derselben Schicht oder, im Fall von strikter Schichtentrennung, auf Schichten direkt unter ihr verweisen kann. Es darf nur eine lose Kopplung geben, die in die entgegengesetzte Richtung zeigt, in Form von injizierten Abhängigkeiten, Callbacks, Observern und so weiter. Diese Regel sorgt für eine gewisse Isolierung zwischen den Schichten und verringert die Kopplung: Änderungen müssen sich im Allgemeinen nur eine Schicht nach oben und/oder unten ausbreiten.

Diese Richtung der Abhängigkeiten bedeutet, dass die höheren Schichten etwas über die darunterliegenden Schichten wissen, aber nicht umgekehrt. Daher liegt es immer in der Verantwortung der unteren Schichten, die Schnittstelle für die Kommunikation zu definieren, und die höhere Schicht muss sich an diese Definition anpassen. Dafür wird dann gern das GoF-Entwurfsmuster Adapter verwendet. Die Schichten sollten jedoch ihre interne Datendarstellung nicht gegenüber höheren Schichten offenlegen, sondern über Datentransferobjekte und Mapper kommunizieren, beides Muster, die von Fowler im Buch „Patterns of Enterprise Application Architecture" näher erläutert werden. Somit schließt sich dann auch der Kreis vom Architekturmuster hin zum Einsatz von Entwurfsmustern.

Die Reihenfolge der Schichten beginnt traditionell mit der Persistenz am unteren Ende, der Geschäftslogik in der Mitte und der Präsentation am oberen Ende. Dies ist eine Folge der Abhängigkeitsregel aus dem Stable Dependency Principle: Weder die Geschäftslogik noch die Präsentation ergeben ohne Daten Sinn, und da Referenzen nur nach unten zeigen können, müssen die Daten ganz unten stehen.

Weiterentwicklung der Schichtenarchitektur

Wer nichts anderes als Applikationen entwickelt, die Daten aus einem Formular in eine Datenbank schreiben, könnte jetzt im Grunde aufhören weiterzulesen. Denn für diese Fälle ist alles bisher Beschriebene völlig ausreichend und die nachfolgend erläuterten Probleme sind keine. Einen Nachteil hat die Drei-Schichten-Architektur nämlich: Sie beschreibt die eigentliche Geschäftslogik nur sehr vage. In besagtem Fall ist das auch nicht so schlimm, weil es da im Grunde keine Geschäftslogik gibt und man als Entwickler eher damit beschäftigt ist, Frameworks aneinanderzustöpseln. Möchte man hier eine mittlere Schicht aufbauen, geschieht das eher mit dem Ziel, alles aus der Präsentations- und Datenschicht herauszulösen, was dort eindeutig nicht hineingehört.



Die Schichtenarchitektur im Überblick (Bild 2)

Bei komplexeren Applikationen sieht dies anders aus. Da haben wir einerseits Geschäftsregeln und Logik, die rein mit der Domäne zusammenhängen. Dinge also, die unabhängig von unserer Anwendung sind. Auf der anderen Seite haben wir anwendungsspezifische Regeln und Workflow-Logik, die nicht mit den Geschäftsregeln unserer Domäne, sondern mit den Aufgaben unserer Anwendung verbunden sind.

Um den Unterschied zu verdeutlichen, betrachten wir eine Anwendung, die Berichte im Finanzwesen erstellt. Dies kann automatisch zu einem bestimmten Zeitpunkt oder manuell durch den Nutzer ausgelöst werden. In einer solchen Anwendung ist die eigentliche Berechnung der Unternehmenseinnahmen ein geschäftliches Anliegen, die Benachrichtigung an Nutzer, dass ein konkreter Bericht erstellt wurde, jedoch eine Workflow-Angelegenheit und keine Geschäftsregel: Wenn der Bericht manuell erstellt wird, müssten die Nutzer nicht über die Erstellung informiert werden, sie haben sie ja selbst ausgelöst. Die Mitteilung an die Nutzer ist also keine geschäftliche Anforderung, sondern einfach die Funktionsweise der Anwendung selbst. Die Verflechtung dieser beiden Verhaltensweisen hat unerwünschte Folgen: Die Zuständigkeiten sind nicht korrekt getrennt, sodass das Single Responsibility Principle verletzt wird, was die Wiederverwendbarkeit unserer Domänenobjekte beeinträchtigt.

Dieses war ausschlaggebend für die Entwicklung der Drei-Schichten-Architektur zur Vier-Schichten-Architektur. Die Geschäftslogikschicht wurde in zwei Schichten aufgeteilt, und zwar in Anlehnung an die Entwurfsmuster Domain Model und Service Layer [4]. Ein Domänenmodell ist ein Netz miteinander verbundener Objekte, wobei jedes Objekt ein sinnvolles Konzept der Domäne darstellt. Seine Belange sind rein geschäftlicher Natur. Alles, was im Domänenmodell geschieht, hat einen geschäftlichen Grund. Das Domänenmodell dient dazu, die Komplexität der Geschäftsdomäne von der technischen Komplexität der Anwendung zu isolieren. Sollte die Geschäftslogik häufigen Änderungen unterworfen sein, wird dieses Domänenmodell auch häufig geändert. Die naheliegendste Möglichkeit, diese Änderungen zu begrenzen, besteht darin, das Domänenmodell so unabhängig wie möglich von den anderen Schichten des Systems zu machen und es somit möglichst stabil zu halten. Dazu bedienen wir uns eines Service Layers beziehungsweise einer Dienstschicht.

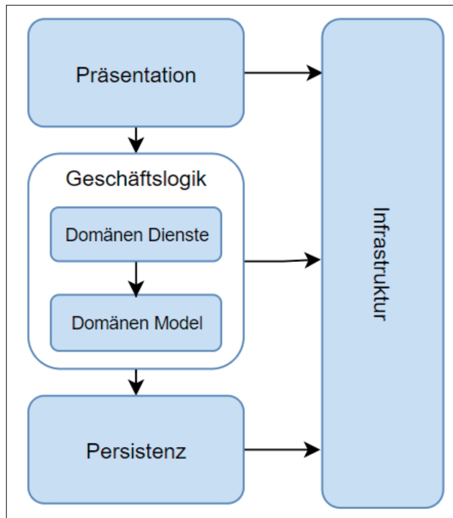
Eine Dienstschicht oder Serviceschicht ist eine Fassade, wie sie von der Gang of Four beschrieben wird, und liegt oberhalb des Domänenmodels. Sie wird geschaffen, um die wesentlichen Geschäftskomplexitäten von der Komplexität der Applikation zu isolieren. Sie bietet einen grobkörnigen Satz von Operationen, über die externe Komponenten (zum Beispiel die Präsentationsschicht) mit der Domäne kommunizieren können. Sie stellt somit eine Abstraktion des Domänenmodells dar und verbirgt dessen Details. Anstatt diese Details also offenzulegen, definiert die Serviceschicht geschäftsorientierte Anwendungsfälle, indem sie Arbeit an Objekte des Domänenmodells delegiert. Auf diese Weise reduziert die Serviceschicht die Kopplung zwischen den höheren Schichten und dem Domänenmodell und kümmert sich auch um die nicht domänenspezifischen, anwendungsbezogenen Anforderungen (technische Aspekte wie Protokollierung, Transaktionsmanagement und so weiter), ohne die Domäne damit zu kontaminieren.

Sollte dies jemanden an Domain Driven Design erinnern, ist das kein Wunder. Eric Evans Buch wurde 2004 veröffentlicht und damit nur zwei Jahre nach „Patterns of Enterprise Application Architecture“ von Martin Fowler. Es erweitert die Idee des von Fowler beschriebenen Domain Models auf ein komplettes Buch, das sich bis hin zu einer ganzen Architekturbewegung entwickelt hat, auf die wir aber später noch zu sprechen kommen werden.

Nachteile der Schichtenarchitektur

Mit der zunehmenden Anzahl von Schichten tritt ein Problem unserer Architektur zutage: Einige übergreifende Belange, wie etwa Messaging, Logging oder Authentifizierung, könnten in verschiedenen Schichten der Anwendung erforderlich sein, doch die Abhängigkeitsregel der Schichtenarchitektur besagt ausdrücklich, dass eine Komponente einer Schicht nur auf andere Komponenten in derselben Schicht oder in der direkten Schicht unter ihr verweisen darf.

Um dieses Problem aufzulösen, müssen wir die Abhängigkeitsregel lockern: Entweder erlauben wir einer Schicht, alle Schichten unter sich zu referenzieren (entspannte Schichtregel) und riskieren damit, Vorteile bezüglich der Kapselung zu verlieren, oder wir schaffen Ausnahmen von der Schichtregel. Diese Ausnahmen sind spezifische Schichten, die von überallher referenziert werden können. Sie stellen Infrastruktur und Dinge wie etwa gemeinsame Basisklassen für bestimmte Klassengruppen zur Verfügung (siehe Bild 3). ▶



Eine Möglichkeit der Vier-Schichten-Architektur mit Querschnittsaspekten (Bild 3)

Dazu gehören häufig Basisklassen wie *ModelBase*, *ServiceBase*, *RepositoryBase* oder *ViewModelBase*.

Während diese Elemente meist in die Infrastrukturebene eingegliedert werden, gehören sie dort eigentlich nicht wirklich hin. Dies kann dazu führen, dass sich ein solches Vorgehen auch auf andere „externe" Elemente ausweitet, die eigentliche Schichtentrennung aushebelt und auf Dauer zu Chaos führt.

Dieses Problem ist leider nicht das einzige, mit dem wir bei der Verwendung der traditionellen Schichtenarchitektur konfrontiert sind. Der wichtigste Nachteil ist die Richtung der Abhängigkeiten. Das Dependency Inversion Principle verlangt ausdrücklich, dass man von Abstraktionen statt von Details abhängig ist. Das Separated Interface Design Pattern von Fowler [4] ist die einfachste Möglichkeit, dies zu tun. Hierbei werden die abstrakten Schnittstellen und deren konkrete Implementierungen auch physisch voneinander getrennt und somit einzeln verpackt, was es ermöglicht, sie unabhängig voneinander zu referenzieren.

Beim Einsatz in der klassischen Schichtenarchitektur führt dies aber zu einem Problem. Jene erfordert, dass die Abhängigkeiten nach unten gerichtet sind, während die getrennten Schnittstellen erfordern, dass die Abhängigkeiten in die entgegengesetzte Richtung zeigen. Aus diesem Grund sind bei der klassischen Schichtenarchitektur die oberen Schichten stark an die darunterliegenden – eher konkreten – Schichten gekoppelt: Es gibt keine einfache Möglichkeit, eine Implementierung durch eine Attrappe oder eine alternative Implementierung für zum Beispiel Tests zu ersetzen. Andererseits wirken sich Änderungen in den unteren Schichten oft auch auf die oberen Schichten aus, weil ebendiese Isolierung fehlt.

Ein weiteres Problem der Schichtenarchitektur besteht darin, dass sie die Applikation entlang ihrer Technologiegrenzen schneidet. Man trennt beispielsweise die Benutzerschnittstellen von der Datenbank. Die schreiende Architektur, die wir erreichen wollen, bezieht sich hingegen auf eine domänenbasierte Trennung der Belange. Denn nur durch diese fachliche Trennung können Anforderungen der Nutzer auch leicht im Code wiedergefunden werden. Durch die klassische Schichtenarchitektur werden Änderungen aber nur selten auf den Auslieferungsgegenstand (englisch: Deployment Unit) abgestimmt: Wenn ein Geschäftsobjekt geändert wird, sind häufig auch Änderungen an seiner Präsentation und Persistenz erforderlich, während diese Belange in verschiedene Schichten verpackt werden.

In Anbetracht all dieser Probleme könnte man sich fragen, ob es nicht etwas Besseres als eine Schichtenarchitektur gibt. Vielleicht sollten wir mehr Wert auf die Umkehrung von Abhängigkeiten und die Externalisierung der Infrastruktur legen, um mehr Flexibilität zu erreichen.

Genau diesen Fragen nähern wir uns im zweiten Teil dieser Serie und erläutern, welche Gemeinsamkeiten so verschiedene Muster wie Ports and Adapters, Zwiebelarchitektur und die Clean Architecture haben. Darüber hinaus gehen wir darauf ein, wie Domain Driven Design sowie CQRS hier hineinspielen und wie sich all diese Muster logisch, aber auch chronologisch ergeben haben. ■

[1] *Frederick P. Brooks, The Mythical Man-Month. Essays on Software Engineering, 1995, Addison-Wesley, ISBN 978-0-201-83595-3*

[2] *Martin Fowler, Refactoring: Wie Sie das Design bestehender Software verbessern, 2020, mitp, ISBN 978-3-95845-941-0*

[3] *Erich Gamma et al., Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software, 2014, mitp, ISBN 978-3-8266-9700-5*

[4] *Martin Fowler, Patterns of Enterprise Application Architecture, 2002, Addison Wesley, ISBN 978-0-321-12742-6*

[5] *Robert C. Martin, Micah Martin, Agile Principles, Patterns, and Practices in C#, 2006, Pearson, ISBN 978-0-13-185725-4*

[6] *Robert C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, 2017, Pearson, ISBN 978-0-13-449416-6*

[7] *Robert C. Martin, The Principles of OOD, www.dotnetpro.de/SL2303CleanArchitecture1*

Hendrik Lösch
ist Consultant und Architekt der ZEISS Digital Innovation. Sein Schwerpunkt liegt auf der Entwicklung und Bewertung von Software auf Basis von Microsoft-Technologien. Darüber hinaus ist er Sprecher und Autor.
hendrik.loesch@zeiss.com

Attila Bertok
arbeitet als Softwareentwickler und -architekt für die ZEISS Digital Innovation Hungary. Er entwickelt komplexe Anwendungen in der Fertigungsindustrie und führt Architekturreviews im DotNet-Technologiestack durch.

dnpCode A2303CleanArchitecture