MODULARE SOFTWARE-ARCHITEKTUR

herbert DOWALIL

2. Auflage

Nachhaltiger Entwurf durch Microservices, Modulithen und SOA 2.0

Mehr Inhalte zum Thema auf der Autorenwebsite 5c-design.info

HANSER

Dowalil

Modulare Softwarearchitektur

Herbert Dowalil

Modulare Softwarearchitektur

Nachhaltiger Entwurf durch Microservices, Modulithen und SOA 2.0

2., überarbeitete Auflage

HANSER

Der Autor:
Herbert Dowalil, Niederösterreich

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Petra Kienle, Fürstenfeldbruck
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Max Kostopoulos
Titelmotiv: © istockphoto.com/myillo
Gesamtherstellung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN: 978-3-446-46377-6
E-Book-ISBN: 978-3-446-46534-3
E-Pub-ISBN: 978-3-446-46631-9

Inhalt

Vorwort

Und jedem Anfang wohnt ein Zauber inne ...
Hermann Hesse

Warum ein Buch zum Thema Modularisierung von Software? Ist dazu nicht bereits alles geschrieben worden? Ist das nicht längst alles klar? Das klappt doch praktisch in allen Softwarehäusern, oder etwa nicht? Was ist also die Motivation eines Autors, gerade darüber ein Buch zu schreiben, und das im 21. Jahrhundert? Um es auf den Punkt zu bringen: Ich glaube keineswegs, dass dieses Thema nirgends mehr Herausforderungen bietet. Ich denke, dass der Grund dafür, dass viele nach wie vor nach Patentrezepten für Softwarearchitektur suchen, genau der ist, dass sie sich fragen, wie man denn eine komplexe Software effizient in ihre Einzelteile zerlegen kann. Der aktuelle Microservice-Hype ist da nur ein Beispiel dafür. Der Wunsch nach einem solchen Allheilmittel für die Softwarearchitektur birgt allerdings ein paar Risiken in sich:

- Man redet aneinander vorbei. Viele haben andere Vorstellungen von den jeweiligen Architekturstilen. Gerade bei SOA und Microservices sind Missverständnisse an der Tagesordnung.
- Das Patentrezept wird angewendet, ohne Rücksicht auf Verluste. Es ist dann kein Mittel zum Zweck mehr, sondern das eigentliche Ziel der Architekturarbeit. Es wird irgendwann gar nicht mehr hinterfragt, ob das Patentrezept für den jeweiligen Anwendungsfall überhaupt sinnvoll ist.

Nach der Lektüre dieses Buchs sehen Sie, lieber Leser, hoffentlich gar nicht mehr den Bedarf nach Patentrezepten, um Ihre Software-Lösung zu strukturieren. Diese sind nämlich gar nicht notwendig, wenn man erstmal die folgenden Dinge verinnerlicht hat:

- Warum man strukturiert, wann man dies besser unterlässt und welche Vor-, aber auch Nachteile die Modularisierung von Software haben kann.
- Welche Möglichkeiten zur Modularisierung bestehen, wann man welche davon anwendet und welche Vor- und Nachteile diese haben.
- Wie man die geeigneten Grenzen für die Modularisierung identifiziert.

Über SOA und Microservices

Die Älteren unter meinen Lesern erinnern sich vielleicht noch an den SOA-Hype!? Jahrelang galt es als hip, genau das zu tun, wenn man eine sehr komplexe Software im Unternehmenskontext entwickelt hatte. Es dauerte eine Weile, bis sich zeigte, dass, aus meiner Sicht und der Meinung mehr oder weniger aller Softwarearchitekten zu dem Thema heute, diese Art von Patentrezept nicht funktioniert. SOA-Projekte scheiterten, was aber nicht zwangsläufig dazu führte, dass man diese auch vorzeitig beendete. Oft sind Organisationen hier nicht in der Lage, sich ein Scheitern einzugestehen. Man hält an dem einmal eingeschlagenen Weg so lange fest, bis sich der Fehlschlag nicht mehr leugnen lässt, weil die Organisation sonst ihr Gesicht verlieren könnte. Was wir heute als Überbleibsel dieses Hypes nach wie vor haben, sind Enterprise-Architekturen voller technischer Schuld, inklusive der deswegen unzufriedenen Auftraggeber und frustrierten Entwickler. Lösungsrezepte als Auswege aus solch verfahrenen Situationen liefert dieses Buch in Kapitel 9.

Aktuell wurde der SOA-Hype vom Microservice-Hype beinahe 1:1 abgelöst, wobei viele Enterprise-Architekten – nicht ganz zu Unrecht – daran zweifeln, ob dieser Architekturstil auch für eine komplexe Unternehmensanwendung brauchbar ist. Viele übersehen dabei, dass, ganz ähnlich wie beim SOA-Hype, nicht klar ist, wie genau denn dieser Architekturstil definiert ist. Ich sehe aktuell verschiedene mögliche Interpretationen, die sich zwar in gewissen Grundzügen ähneln, aber doch auf verschiedene Ziele hin optimiert sind. Ich werde auf diese im weiteren Verlauf des Buchs natürlich eingehen und auch erklären, warum der SOA-Hype gescheitert ist.

Womit wir bei einem leidigen Thema wären, welches oft Debatten im Bereich der Softwarearchitektur schwierig macht, nämlich der Art und Weise, wie wir Fachbegriffe benutzen. Anders als Sparten wie Medizin oder Biologie haben wir es bisher versäumt, eine allgemein anerkannte Begriffswelt zu definieren. Inzwischen gibt es einen beachtenswerten Versuch des iSAQB, wo es allerdings Erfolg und Verbreitungsgrad noch abzuwarten gilt. Solange wir eine solche gemeinsame Sprache nicht haben, bleibt auch mir als Autor nichts anderes übrig, als die Begriffe, die ich in diesem Buch verwende, explizit zu definieren. Dem Thema habe ich daher am Ende dieses Buchs einen Abschnitt gewidmet.

Apropos Anhang des Buchs. Da viele der Muster und Prinzipien, die wir in diesem Buch besprechen, in mehr als einem Abschnitt diskutiert werden, habe ich diese mit eindeutigen Identifiern versehen, über die ich mich immer wieder darauf beziehe, wie P02-SoC für das Prinzip „Separation of Concerns“. Diese sind im Anhang ebenfalls nochmal zusammengefasst und außerdem in Relation zueinander gestellt. Dasselbe gilt für diverse Antipattern (A??-) und Gesetze wie Conway’s Law (L??-), die dort ebenfalls angeführt sind. Darüber hinaus finden Sie dort noch klassische Trade-offs (T??-) der Softwarearchitektur. Dabei handelt es sich um Zwickmühlen, aus denen es keinen klaren Ausweg gibt. Im konkreten Fall muss man jeweils abwägen, für welche der Optionen man sich entscheidet.

Was ich aber zuerst klären möchte, u. a. weil eine klare und allgemeine Definition fehlt, ist, was ich unter dem Begriff Softwarearchitektur verstehe und worum es genau in diesem Buch eigentlich gehen wird. Diesem Thema widme ich gleich im Anschluss das erste Kapitel.

Als roter Faden durch dieses Werk wird sich ein Tutorial zum Thema Modularisierung von Code ziehen. Die Beispiele dafür (genannt „Labs“) finden Sie auf der Website des Verlags für Java (*https://downloads.hanser.de/*). Es gibt außerdem noch eine etwas veraltete Version

davon in C# auf github unter *https://github.com/hdowalil/software-design-c_sharp* zu finden. Diese basiert noch auf .NET Core 2.0, was aber auf den Code selbst keinen Einfluss hat. Der C#-Code wird von mir nicht mehr gewartet, ist aber, wenn Sie C# der Sprache Java deutlich bevorzugen, durchaus brauchbar.

Um die Labs auf Ihrem Rechner nachvollziehen zu können, benötigen Sie ein JDK (mindestens Version 9) und eine IDE, die Java-Projekte kompilieren kann, die auf Maven basieren. Sehr gut klappt das mit der „Eclipse IDE for Java Developers“, in der das m2e-Plugin für die Maven-Integration in Eclipse bereits inkludiert ist. Einfach das .zip-File lokal entpacken und in Eclipse beide Projekte (training-clean und training-design) mit „Import Project“ und danach „Existing Maven Project“ importieren.

Der Aufbau der einzelnen Labs ist dabei immer derselbe. Es gibt jeweils ein Negativ- („Challenge“) und ein Positiv-Beispiel („Solution“), welche auch in diesem Buch diskutiert werden („Sample“). Dazu noch eine weitere Übung („Excercise“), an der Sie sich selbst versuchen können, um im Endeffekt Ihre Lösung („Solution/Your“) mit der Vorgabe des Autors („Solution/Trainer“) zu vergleichen. Dabei wünsche ich Ihnen viel Spaß!

Die Notation der Abbildungen

Mir sind die Vorteile eines Standards wie der Unified Modelling Language (UML) natürlich vollkommen bewusst. Ich habe auch für die 2. Auflage meines Buchs beschlossen, explizit (mit wenigen Ausnahmen) nicht auf diese zu setzen. Meiner Erfahrung nach bringt die enorme Komplexität der UML auch gewisse Gefahren mit sich. Wussten Sie beispielsweise, dass unterschiedliche Pfeilspitzen-Darstellungen im Sequenzdiagramm bedeuten, dass die jeweilige Kommunikation synchron oder asynchron passiert? In der Theorie bietet einem die UML den Vorteil, dass sie ungemein aussagekräftig ist, allerdings nur, wenn jedem, der die Diagramme bearbeitet und liest, auch all diese Feinheiten bewusst sind. Wenn also jemand synchrone und asynchrone Kommunikation im Diagramm festhält, sollte der Leser, denselben Wissensstand vorausgesetzt, genauestens Bescheid wissen. In der Praxis erlebe ich aber oft, dass es genau deswegen zu Missverständnissen kommt. Entweder wusste der Autor des Diagramms nichts darüber und das Diagramm wird vom Leser fehlinterpretiert, oder aber der Leser übersieht diese möglicherweise wichtige Information. Der Nachteil, wenn man nicht auf einen Standard wie die UML setzt, ist der, dass man die eigene Notation erklären muss. Ich mache das vor allem, was die Bedeutung der Pfeilrichtungen angeht: Es geht in meinen Diagrammen, wenn diese nicht auf der UML basieren, meist um statische Abhängigkeit, nicht etwa um zeitliche Reihenfolge oder Datenflüsse. Nur wenn die Pfeile auch mit Zahlen notiert sind, geht es um den Ablauf einer Interaktionssequenz.

Danksagung

Danken möchte ich allen, die mich zu diesem Buch inspiriert und ermutigt haben, sowie meiner Frau und meiner Tochter, die mir den Freiraum dafür geschaffen und es mir ermöglicht haben. Inspiriert zu den Inhalten wurde ich von aktuellen und ehemaligen Kollegen wie Stefan Toth, Stefan Zörner, Peter Götz, Rene Weiss, Oliver Zeigemann, Mohammad

Kabiri, Michael Koitz, Miroslaw Szelag, Christa Dihanich, Jürgen Kofler, Milan Heimschild und Sebastian Bicchi. Ermutigt, meine Inhalte als Buch zu veröffentlichen, haben mich in erster Linie Gernot Starke und Alexander von Zitzewitz. Danken möchte ich aber auch jenen, die mit einer entweder hoffnungslos veralteten Einstellung oder einer bizarren Überzeugung zum Thema Softwarearchitektur meinen Skill zur Argumentation zu diesem Thema geschärft haben. Letztere werde ich an dieser Stelle allerdings nicht namentlich anführen.

Der Autor

Herbert Dowalil, Jahrgang 1976, ist glücklich verheiratet und lebt mit seiner Frau und der gemeinsamen Tochter im Großraum Wien. Er begann bereits zu seiner Grundschulzeit, in den 80er-Jahren des vergangenen Jahrhunderts, sich selbst das Programmieren beizubringen. Dabei begann er mit Acorn Electron Basic, um dann Turbo Pascal, C und C++ und später im 21. Jahrhundert Java zu verwenden. Dabei legte er sein Hauptaugenmerk bald auf die Frage, was gut wartbare und robuste Systeme von den Problemfällen, welche man in der Branche ja leider zu oft antrifft, unterscheidet. Heute beschäftigt er sich mit Themen wie Vermessung von Software in Kennzahlen, Mikro- bzw. Makro-Architekturen, Design Pattern und Prinzipien des modularen Softwareentwurfs. Spezialisiert ist er u. a. auf Refactorings und Evolution von Legacy-Architekturen.

Kontakt: *hdowalil@gmail.com*

Im World Wide Web:

https://5c-design.info

Trainingsvideos passend zu den Inhalten dieses Buchs gibt es hier:

https://www.udemy.com/user/herbert-dowalil/

1 Über Softwarearchitektur

Your mind is just a program and I am the virus. I'm changing the station.
I will improve your thresholds.
Muse

1.1 Die Softwarekrise

Begonnen hat alles im Jahr 1968, als ein gewisser Edsger W. Dijkstra ein Papier mit dem Titel „A Case against the GO TO Statement" [Dij68] veröffentlichte. Warum war dies damals notwendig? Hardware wurde immer leistungsfähiger. Als die ersten Programme geschrieben wurden, war es kaum nötig, sich über Dinge wie Wartbarkeit Gedanken zu machen. Die Hardware war erstmals schlichtweg nicht ausreichend leistungsfähig für Programme, welche komplex genug waren, um nicht mehr verstanden zu werden. Im Laufe der 60er-Jahre des 20. Jahrhunderts änderte sich dies aber schön langsam. Man bemerkte erstmals, dass Code, der von einem Programmierer entwickelt wurde, von einem anderen kaum noch verstanden werden konnte. Man bezeichnete diese Problematik der immer komplexer werdenden Software als die „Softwarekrise". Als Dijkstra sein Papier veröffentlichte, in dem er dafür plädierte, auf Sprunganweisungen (GOTO) im Code zu verzichten und stattdessen auf Schleifen (for, while) zu setzen, sorgte dies zunächst für heftige Diskussionen, die aus heutiger Sicht vermutlich nicht mehr nachvollzogen werden können. Heute gilt es unter Softwareentwicklern klar als Konsens, dass man auf die Verwendung von GOTO-Sprüngen verzichten sollte.

Um dies zu verstehen, bitte ich Sie, einen Blick auf Bild 1.1 zu werfen. Die beiden Blöcke links und rechts repräsentieren einen vom Ergebnis her äquivalenten Programmablauf, wobei im linken Sprünge zur Anwendung kommen, im rechten aber nicht. Tatsächlich ist der rechte Ablauf für einen Menschen einfacher zu verstehen, während dies natürlich für einen Compiler völlig irrelevant ist. Woran liegt das? Das menschliche Gehirn begreift einen komplexen Sachverhalt, indem es diesen in seine Einzelteile, sogenannte „Chunks" (Bündel), zerlegt. Von Code, der aber von x-beliebigen Sprüngen durchzogen ist, lässt sich nicht gut ein gedanklich „gebündeltes" Modell erzeugen, das einem dabei hilft, ihn zu verstehen und den Überblick zu behalten.

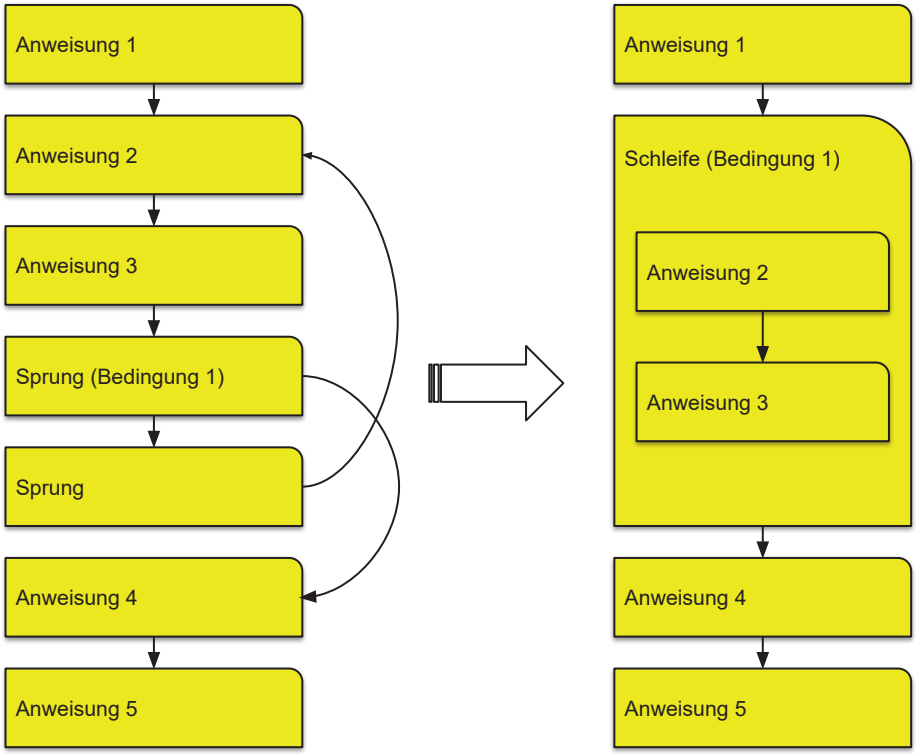

Bild 1.1 Sprünge im Code vs. Schleifen

Laut einer durchaus anerkannten Theorie eines gewissen George A. Miller [Mil55] tun wir uns als Menschen außerdem schwer damit, mehr als ca. sieben (zwischen minimal fünf bis maximal neun) Chunks im Kopf zu behalten. Was aber tun, wenn der Sachverhalt komplexer ist als das? Wenn er aus mehr Einzelteilen besteht, welche immer noch verstanden werden müssen? Wir helfen uns dann damit, dass wir derart komplexe Systeme in Substrukturen aufteilen. Das Verstehen selbst läuft dann auf unterschiedlichen Ebenen der Abstraktion. Zur Verdeutlichung möchte ich ein Beispiel bringen, welches in Zusammenhang mit einem meiner Lieblingshobbies steht, nämlich dem Bergsteigen. Angenommen ich möchte meinen Lieblingsklettersteig in den Wiener Voralpen gehen – den Teufelsbadstubensteig auf die Rax –, dann überlege ich mir zunächst einmal, wie ich über die Autobahn in die Gegend der Wiener Hausberge komme. Dafür fahre ich von der Wiener Neustädter Gegend über die Autobahnen A2 und S6 bis Gloggnitz. Sobald ich dort abgefahren bin, orientiere ich mich etwas lokaler und fahre zum Parkplatz in der Weichtalklamm. Sobald ich das Klettersteigset angelegt und den Rucksack umgeschnallt habe, orientiere ich mich wiederum noch lokaler (also auf einer weiter unten liegenden Abstraktionsebene) und bewege mich auf den markierten Wanderwegen zum Einstieg in den Teufelsbadstubensteig. Dabei haben wir uns in Summe auf drei Abstraktionsebenen bewegt, einerseits bei der Fahrt von einem Bezirk in den anderen über zwei Autobahnen, dann entlang der Bundesstraße zum Parkplatz und schließlich zu Fuß zum Einstieg in den Klettersteig. Tatsächlich gibt es darüberliegend noch weitere mögliche Abstraktionsebenen. Bei einer Dienstreise

nach München sind mir die einzelnen Bezirke, die ich durchquere, egal und ich navigiere von einem Land zum anderen. Wenn ich geopolitische Zusammenhänge verstehen möchte, denke ich auf einer noch höheren Ebene der Großmächte und Kontinente.

Wenn wir dies nun auf die Welt der Softwarearchitektur übertragen, stellt sich die Frage, was man tun soll, wenn einfache Programmstrukturen nicht ausreichen, einfach weil man mehrere solcher Ebenen benötigt? Die Lösung besteht auch hier darin, Strukturen auf höheren Abstraktionsebenen zu bauen. Siehe dazu Bild 1.2, wo Baustein 1 und Baustein 2 miteinander über eine Schnittstelle interagieren, die Baustein 1 anbietet. Baustein 1 selbst zerlegt sich aber wiederum in die Bausteine A, B und C. Man fasst also Programmstrukturen in Methoden zusammen, diese bilden gemeinsam Klassen, welche in Modulen gebündelt werden. Für dieses Spiel bieten die einzelnen Technologieplattformen unterschiedliche Möglichkeiten. So gibt es in Java über den Klassen Packages, welche sich wiederum zu JPMS-Modulen zusammenfassen lassen. Dieses Spiel, auf so viele Abstraktionsebenen wie nötig gespielt, ist es, was Software dauerhaft effizient wartbar macht. Und genau davon handelt dieses Buch.

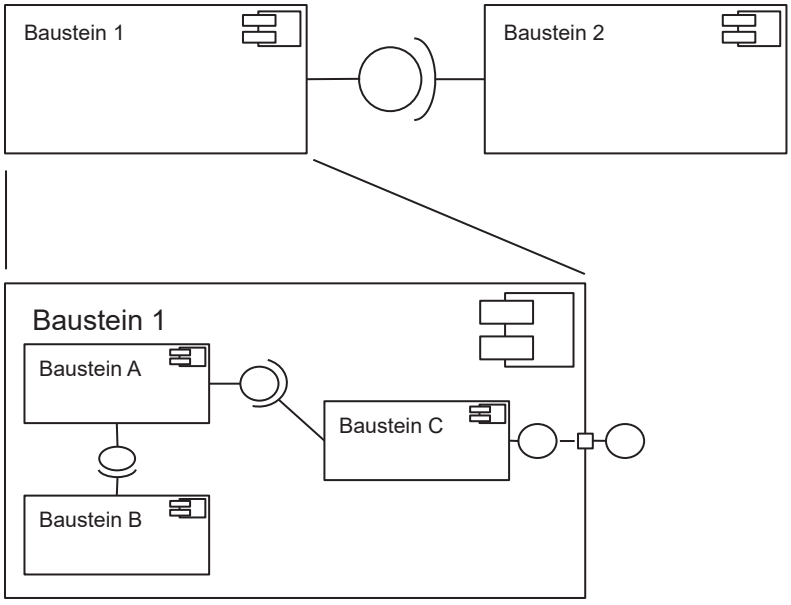

Bild 1.2 Eine in der UML dargestellte hierarchische Zerlegung

Ein Diagramm wie das in Bild 1.2 bietet eine rein statische Sicht auf das Zusammenspiel von Bausteinen. In der Black-Box-Sicht sehen wir erst mal nur die außerhalb wahrnehmbaren Eigenschaften eines Bausteins, welche Aufgabe er erfüllt und welche Schnittstellen er im Sinne eines Vertrags nach außen anbietet. Wenn wir einen Blick in einen dieser Bausteine hinein werfen, so nennt man dies eine White-Box-Sicht, die den inneren Aufbau eines Bausteins zeigt. Ein Baustein sollte dabei, wenn sauber entworfen, die folgenden Eigenschaften aufweisen:

- Einzelne Bausteine können möglichst ohne Seiteneffekte auf andere Bausteine und Abstimmungsarbeit mit anderen Teams weiterentwickelt werden.
- Einzelne Bausteine sind unabhängig vom Rest zu dokumentieren und zu verstehen.
- Einzelne Bausteine sind an ihren ein- und ausgehenden Schnittstellen isoliert zu testen.
- Einzelne Bausteine können unabhängig voneinander ausgetauscht werden.

Diese statische Sicht auf die einzelnen Bausteine ist allerdings nicht die ganze Wahrheit. Man sieht darin nicht, ob, wann und wie oft eine solche Interaktion an einer der Schnittstellen erfolgt. Deshalb kann es auch interessant sein, auch die dynamische Sicht eines solchen Zusammenspiels zu zeigen. Dafür wiederum eignet sich ein Sequenzdiagramm der UML wunderbar, welches vereinfacht in Bild 1.3 für den Ablauf des sogenannten Circuit-Breaker-Patterns dargestellt ist.

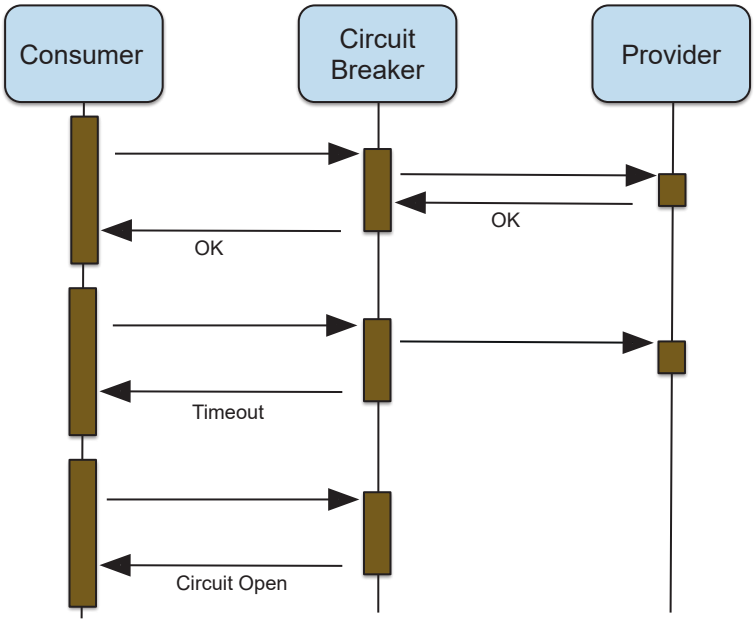

Bild 1.3 Der Ablauf des Circuit-Breaker-Patterns, in Anlehnung an die UML

1.2 Was ist Softwarearchitektur überhaupt?

Was ist nun eigentlich Softwarearchitektur? Eine allgemein anerkannte und wirklich von allen gleichermaßen benutzte Definition dafür gibt es nicht. Man muss wohl unterscheiden zwischen verschiedenen klassischen Definitionen, bei denen strukturelle Aspekte von Software im Vordergrund stehen. Dabei geht es darum, wie komplexe Software in ihre Substrukturen zerlegt wird, und um das Zusammenspiel ihrer einzelnen Bausteine. Das Problem bei dieser Definition ist allerdings, dass es meist nicht weit genug geht. Einerseits gibt

es Themen, welche mit der Zerlegung in einer so engen Wechselwirkung stehen, dass sie davon schlecht getrennt werden können. Andererseits ist die Zerlegung in Substrukturen kein Ziel mit Selbstzweck, sondern dazu da, langfristig Wartbarkeit sicherzustellen. Womit wir beim Thema Qualitätsmerkmale von Software wären und somit bei den Kompromissen, welche man bei deren Erfüllung als Architekt oft eingehen muss. Betrachten wir mit dem Standard ISO 9126 [WIK19-a] einen der bekannteren Standards zum Thema Softwarequalität. Dieser legt folgende Kategorien und Unterkategorien fest:

Änderbarkeit/Wartbarkeit

Dabei wird die Frage beantwortet, welche Aufwände durch bestimmte Änderungen an einer Software verursacht werden. Unterkategorien sind:

- Analysierbarkeit
- Modifizierbarkeit
- Stabilität
- Testbarkeit

Effizienz

Hier geht es um das Verhältnis der Leistung der Software und der dafür nötigen Ressourcen. Auch hier gibt es Unterkategorien:

- Zeitverhalten
- Verbrauchsverhalten

Funktonalität

In welchem Ausmaß ist die geforderte Funktionalität auch korrekt? Von einer Telebanking-Software wird wohl immer eine korrekte Antwort verlangt, wenn diese vielleicht manchmal auch nicht ganz aktuell ist. Bei einer Spracherkennung mag eine korrekte Erkennung des Sprachinputs in 98 % der Fälle genügen, während es weniger akzeptabel scheint, wenn eine Verarbeitung erst an einem späteren Tag erfolgt. Unterkategorien:

- Angemessenheit
- Sicherheit
- Interoperabilität
- Ordnungsmäßigkeit
- Richtigkeit

Übertragbarkeit

Dabei geht es darum, ob bzw. mit welchem Aufwand die Möglichkeit gegeben ist, eine Software von einer Umgebung in eine andere zu übertragen. Unterkategorien hierfür sind:

- Anpassbarkeit
- Austauschbarkeit
- Installierbarkeit
- Koexistenz

Zuverlässigkeit

Welches Leistungsniveau bietet die Software über welchen Zeitraum unter welchen festzulegenden Bedingungen? Unterkategorien sind:

- Fehlertoleranz
- Reife
- Wiederherstellbarkeit

Benutzbarkeit

Individuelle, teils subjektive Bewertung durch die Benutzer und Aufwand, der für diese nötig ist, um die Software zu erlernen bzw. zu bedienen. Unterkategorien:

- Attraktivität
- Bedienbarkeit
- Erlernbarkeit
- Verständlichkeit

Der letzte Punkt Bedienbarkeit verdient noch mal besondere Beachtung. Man kann nämlich – in den Unterkategorien ersichtlich – Verschiedenes darunter verstehen. Bedienbarkeit ist ein typischer Wunsch von Sachbearbeitern, die sehr oft mit derselben Software arbeiten. Dafür darf durchaus etwas spezielles Know-how vonnöten sein, das man sich auf Dauer aneignet. So besteht hier der Wunsch, gewisse Dinge über Hotkeys abkürzen zu können. Oft gibt es sehr viele Optionen auf einzelnen Bildschirmmasken, welche nur der Poweruser noch überblicken und bedienen kann. Wenn Sie allerdings Benutzer über das Web auf Ihre Plattform locken möchten, damit sich diese dort registrieren, so werden Sie diesen wohl eher etwas mehr Führung und Übersicht anbieten wollen. Dieser Wunsch nach einfacher Erlernbarkeit steht demnach durchaus im Widerspruch zum Wunsch der Bedienbarkeit. Wir sehen also: Einzelne geforderte Qualitätsmerkmale einer Software können im Widerspruch zueinander stehen. Beim Entwurf einer Softwarearchitektur ist es schlichtweg so, dass sich nicht immer alle Wünsche unter einen Hut bringen lassen (T01-Quality). Außerdem kollidieren Wünsche nach bestimmten Qualitäten auch nicht selten mit technischen oder organisatorischen Rahmenbedingungen in Projekten. Bei solchen Rahmenbedingungen handelt es sich um Dinge wie das zur Verfügung stehende Budget.

Bei der Modularisierung von Software geht es in erster Linie um die Qualitätsmerkmale der Kategorie Änderbarkeit bzw. Wartbarkeit. Auch diese können im Widerspruch stehen, und zwar beispielsweise zu Zielen der Effizienz. Um wartbar zu sein, empfiehlt es sich, Strukturen zu bauen, Schichten, Adapter, Module und explizite Schnittstellen zu entwerfen. Dies kann aber wiederum Systemressourcen (wie CPU) verbrauchen. In manchen Fällen mag dies relevant sein, wie beispielsweise bei der Entwicklung von Embedded-Systems oder wenn Sie einen Schachalgorithmus schreiben möchten, der AlphaZero Konkurrenz machen soll. In den allermeisten Fällen führt aber kein Weg um wenigstens ein gewisses Mindestmaß an Wartbarkeit durch Modularisierung herum. Ich habe schon Teams erlebt, wo dies explizit niedrig priorisiert wurde, weil der Plan ohnehin war, die Software nur für einen gewissen Zeitraum einzusetzen, die dann aber während der erstmaligen Entwicklung dieser bereits in Probleme gekommen sind. Abgesehen davon habe ich aus Erfahrung gelernt, dass Software, deren Einsatz nur für kurze Zeit geplant wird, im Endeffekt doch länger lebt als gedacht.

Was ich zeigen möchte, ist, dass man das Thema Softwarequalität nicht gut vom Thema Modularisierung trennen kann. Wenn Sie eine etwas weiter gefasste Definition von Softwarearchitektur haben möchten, dann muss es bei dieser generell um wichtige Entscheidungen gehen, die bei der Erstellung und Weiterentwicklung von Software getroffen werden. Diese stehen dann meist in Zusammenhang mit den geforderten Qualitäten der Software. Wichtig ist eine Entscheidung dann, wenn sie Einfluss auf die Erfüllung der Qualitätsziele hat, sich stark auf den Entwurf der Software auswirkt oder sich später nicht mehr einfach ändern lässt.

Falls Sie der Meinung sind, dass die Ermittlung der geforderten Qualitäten einer Software beim Kunden Aufgabe des Requirement-Engineerings wäre, dann muss ich Sie enttäuschen. Auch wenn es das theoretisch ist, so sind es doch wir Softwarearchitekten, die diesen Input mehr oder weniger als einzige benötigen, weshalb es im Endeffekt auch an uns liegt, diesen zu erheben. Das Gute daran ist, dass man den Kunden dabei direkt auf Kompromisse aufmerksam machen kann, die dabei gemacht werden müssen.

In diesem Jahrtausend hat sich am grundlegenden Verständnis von Softwarearchitektur übrigens kaum mehr etwas geändert. Große Paradigmenwechsel auf dieser ganz allgemeinen Ebene gab es (noch) nicht. Falls es manchmal dennoch so scheint, so handelt es sich um Hypes, wie Microservices, die dann teilweise als „neue Softwarearchitektur“ verstanden werden, aber im Grunde nichts weiter sind als neue Architekturstile. Wenn auch, wie im konkreten Fall der Microservices, recht interessante.

Um es noch mal auf den Punkt zu bringen: Auch wenn ich hier eine etwas weitere Definition des Begriffs Softwarearchitektur biete, so wird es in diesem Buch aber um das Thema Änderbarkeit bzw. Wartbarkeit durch effiziente Strukturierung bzw. Modularisierung gehen. Von den untersten Abstraktionsebenen des Codes bis hinauf zur Enterprise-Architektur, für welche ich in diesem Werk moderne und effiziente Ansätze liefern werde (Kapitel 8). Wenn Sie an einem guten Buch interessiert sind, welches allgemeiner auf die weitere Definition von Softwarearchitektur eingeht, lassen Sie mich Ihnen das Werk von Gernot Starke [Sta15] empfehlen.

Qualitäten und Änderbarkeit

Noch etwas: Auch wenn Sie eine Software von wunderbar effizienter Struktur entwerfen, so bezieht sich die damit erreichte Änderbarkeit meist auf neue Funktionen, neue Felder, Entitäten und Bildschirmmasken. Neue Qualitätsaspekte dagegen lassen sich üblicherweise nicht ohne großen Aufwand hinzufügen. Ein neues Feature wie „Skalierbarkeit“, „Performance“ oder „Ressourceneffizienz“ ist selten einfach, schnell und billig in eine bestehende Softwarelösung einzubauen. Dies liegt dann aber nicht daran, dass die gewählte Strategie zur Modularisierung nicht effizient wäre, sondern liegt in der Natur der Sache.

Aber: Ohne ein gewisses Niveau an Änderbarkeit wird eine nachträgliche Optimierung einer Software bezüglich eines dieser Qualitätsmerkmale ziemlich sicher scheitern. Wenn jede Änderung ein riskantes Abenteuer darstellt, das unvorhergesehene und destabilisierende Seiteneffekte auslöst, wird ein solches Unterfangen ziemlich sicher scheitern.

1.2.1 Wartbarkeit von Code

Was zeichnet nun besonders gut wartbaren Code aus? Bzw. was genau bedeutet es eigentlich, wartbar zu sein? Es kann sich dabei um unterschiedliche Aspekte handeln wie:

- Die Einarbeitungszeit für neue Entwickler ist vergleichsweise kurz. Dadurch können neue Mitarbeiter relativ günstig an Bord geholt werden, um danach produktiv mitzuarbeiten.
- Der Aufwand zur Identifikation der von der gewünschten Änderung betroffenen Codeteile hält sich in Grenzen.
- Das Verständnis des zu ändernden Codes ist vergleichsweise einfach möglich. Der Code hält sich an die gängigen Konventionen, die die allermeisten Entwickler kennen. Der Code ist außerdem selbsterklärend oder inline dokumentiert.
- Änderungen können den bestehenden Konzepten entsprechend effizient umgesetzt werden. Für eine gewünschte Änderung müssen bestehende Konzepte möglichst nicht adaptiert werden. Neuerungen sind dem Code wiederum relativ einfach hinzuzufügen, ohne dass der bestehende Code an vielen Stellen dafür geändert werden muss.
- Die Entwickler verstehen die Auswirkungen der Änderungen und das Risiko unerwünschter Auswirkungen hält sich in Grenzen.
- Implementierte Änderungen entsprechen danach auch den tatsächlichen Wünschen des Auftraggebers.
- Die Neuerungen bzw. Änderungen lassen sich vergleichsweise einfach auf ihre Fehlerfreiheit hin überprüfen. Genau genommen gilt dies auch für die bestehenden Features, wo einfach festgestellt werden kann, ob deren Qualität unter den Neuerungen gelitten hat.
- Der Release der Änderungen, sodass diese auch beim Kunden wirksam werden, ist relativ flott und problemlos möglich.

In diesem Buch wird es in erster Linie darum gehen, wie man durch Strukturierung und Modularisierung wartbare Systeme entwickelt. Auf Code-Ebene gibt es teilweise unterschiedliche Strategien, die jeweils andere der o. a. Aspekte der Wartbarkeit fördern. Das folgende Beispiel soll dies verdeutlichen. Als ich eine frühe Version meiner App „Finde den Käse“ (siehe: *https://play.google.com/store/apps/details?id=net.dowalil.findthecheese*) meiner Tochter zeigte, hatte sie natürlich sofort Ideen, was man noch alles hinzufügen könnte. In dieser App führt man eine Maus durch Programmierung vier einfacher Befehle zum Käse (Bild 1.4). Sie meinte dann, dass man jetzt noch eine Katze programmieren könnte, die versuchen würde, die Maus zu fressen. Außerdem Steine, die verschoben werden können, ein Trampolin, mit dem sie über die Mauer springen könnte, ein Mauseloch und so weiter. Angenommen, wir beschließen, so wie ich es tatsächlich getan habe, die Katze zu implementieren, was haben wir dann für Möglichkeiten und wie wirken sich diese später auf die Wartbarkeit der Software aus? Wir wollen ja auf keinen Fall ausschließen, später noch einige der anderen Ideen zu implementieren. Es gab dafür zwei Möglichkeiten:

- Man schreibt den Algorithmus für die Katze nach allen Regeln des Clean Codes. Man denkt dabei gar nicht an weitere Entwicklungen und folgt dem „You-ain't-gonna-need-it“-Prinzip (P01-YAGNI), nachdem man keine in Zukunft möglichen Änderungen vor-

wegnehmen sollte. Die Lösung ist nicht generisch und dadurch für andere Entwickler einfach zu verstehen. Andererseits wäre es relativ aufwendig, eine neue Figur hinzuzufügen

- Die Alternative wäre eine abstrakte Klasse „Spielfigur“, die sich durch Festlegung der Verhaltensregeln individualisieren lässt. Diese Lösung wäre relativ komplex, aber einfach zu erweitern.

Bild 1.4 Level 6 aus „Finde den Käse“

Beide Lösungen sind valide. Man kann in so einem Fall überlegen, ob öfter neue Spielfiguren entwickelt oder neue Mitarbeiter eingeschult werden. Tendenziell empfehle ich aber, dem YAGNI-Prinzip zu folgen und auf die Vorwegnahme von Komplexität zu verzichten. Der Kreativität meiner Tochter sind schließlich keine Grenzen gesetzt, und wenn sie sich einmal eine variable Spielfeldgröße wünscht, so wäre die Komplexität der abstrakten Spielfigur dabei gar nicht hilfreich. Vielleicht sogar im Gegenteil, wenn der komplexe und generische Code dann angepasst werden muss an den dann doch überraschenden neuen Wunsch.