Neue Serie: Modulithen S. 44

Agile Architektur für Unternehmen S. 68

Terminalbibliotheken im Vergleich S. 76

JAVA Mag

Java magazin™

Java | Architektur | Software-Innovation

JAVA 15

GRUNDSTEINLEGUNG FÜR DIE ZUKUNFT

Ausgabe 12.2020

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 12

www.javamagazin.de

Architekturpatterns in Modulithen – Teil 1

Ordnung ins Chaos bringen

„Wir haben diesen Legacy-Monolithen, den wollen wir in Microservices aufbrechen“. So einen Satz hört man als Berater in der Softwarebranche oft. Auf die Frage „Warum“ erhält man oft die Antwort „Modularisierung“. Denn es herrscht die weitverbreitete Ansicht, dass Monolithen grundsätzlich aus schlecht strukturiertem Legacy-Code bestehen und sich Monolithen und Modularisierung gegenseitig ausschließen. Dass dem nicht so ist, zeigt die Architekturform der Modulithen. In dieser Artikelserie wird sie beleuchtet und beschrieben, mit welchen Patterns ein Modulith gelingen kann und welche Antipatterns man dabei vermeiden sollte.

von Arnold Franke

Wer in den letzten sieben Jahren Softwarekonferenzen besucht hat, der musste ganz schön Slalom laufen, wenn er dem Thema Microservices [1] aus dem Weg gehen wollte. Aufgrund ihrer vielversprechenden Eigenschaften wurde diese Form der Systemarchitektur stark gehypt und von vielen Entwicklern, Architekten und Firmen als Heilsbringer angesehen. Mit etwas Abstand betrachtet, handelt es sich dabei aber um keine Silver Bullet, sondern nur um eine von vielen Möglichkeiten, ein Softwaresystem zu strukturieren – mit eigenen Vor- und Nachteilen (**Abb. 1**). Eine Landschaft aus wirklich entkoppelten Microservices zu bauen, die alle Vorteile dieser Architekturform mitnimmt, ist alles andere als trivial, und man ist selbst dann nicht gegen strukturelle Stolperfallen wie Conway's Law [2] gefeit. Wenn man Microservices ungünstig schneidet und verknüpft, dann besteht genauso die Gefahr, am Ende einen großen Deployment-Monolithen aus Legacy Code zu erhalten.

Den Hype überlebt haben die „guten alten“ Monolithen. Große Deployment-Einheiten mit einer riesigen Codebase, die sich den Ruf von schlechter Wartbarkeit

und Erweiterbarkeit eingehandelt haben. Einerseits überlebten sie, weil man einige davon aufgrund gewachsener Komplexität und schlecht auflösbarer Abhängigkeiten nur schwer wieder losbekommt. Andererseits, weil sie in Form der modularen Monolithen – Modulithen – eine Renaissance in der Entwicklercommunity erleben. Modulithen vereinigen einige der Vorteile der Microservices-Architektur, zum Beispiel modulare Struktur mit gekapselten Verantwortlichkeiten und übersichtlichen Abhängigkeiten, während sie auf einige der Nachteile, wie beispielsweise komplexe Infrastruktur und Kommunikations-Overhead, verzichten. Das macht die Modulithen nicht zu einer den Microservices überlegenen Architekturform (Kasten: „Was können Microservices, was ein Modulith auch kann?“). Sie sind nur ein weiterer valider Ansatz zur Strukturierung eines Softwaresystems, der in bestimmten Kontexten sinnvoller ist als andere.

Die Wahl der Architekturform hängt letztendlich davon ab, auf welchen Aspekt man optimieren möchte. Individuelle Skalierung? Entwicklungsgeschwindigkeit? Resilience? Komplexität der Infrastruktur? Das sind alles Faktoren, die bei dieser Entscheidung eine Rolle spielen.

Dabei ist die Idee des modularen Monolithen durchaus keine neue. Sie ist aber nur erfolgreich und nachhaltig umsetzbar, wenn man es schafft, durch nachhaltige Anwendungsarchitektur die Komplexität einer großen Codebase im Zaum zu halten und Verständlichkeit und Wartbarkeit des Codes zu wahren. Damit habe ich mich in den letzten sieben Jahren in mehreren Softwareprojekten beschäftigt und möchte in dieser Serie einige Patterns und Antipatterns teilen, um anderen Modulithen-Bauern leichter zum Erfolg zu verhelfen.

Den Monolithen schneiden

Egal ob man einen historisch gewachsenen Monolithen besser strukturieren will oder ein großes Projekt auf der grünen Wiese startet – eine der ersten Entscheidungen, die man treffen muss, ist: „Wie schneide ich meinen Code in Module?“ Dazu muss man sich bewusst machen, was ein gutes Modul ausmacht: Einer der wichtigsten Aspekte einer modularen Architektur ist die Separation of Concerns. Ein Modul hat eine klare Verantwortlichkeit und ist nur für diese zuständig. Es kümmert sich dabei nicht um Verantwortlichkeiten anderer Module (kein Featureneid [3]). Es kapselt seine Verantwortlichkeit und alle dazu benötigten internen Aspekte wie die Implementierung von Businesslogik und die Persistenz von Daten, sodass sie nicht von außen manipulierbar sind.

Um mit der Außenwelt und anderen Modulen zu kommunizieren, stellt ein Modul Schnittstellen für die notwendigen Operationen bereit, die kontrollierten Zugriff auf seine Verantwortlichkeit erlauben. Es ist möglichst entkoppelt von anderen Modulen und hat so wenige Abhängigkeiten wie möglich. Intern weist es eine hohe Kohäsion auf, d. h., dass es keine „Inseln“ enthält, die mit dem Rest des Moduls nichts zu tun haben.

Abb. 1: Vier Jahre später …

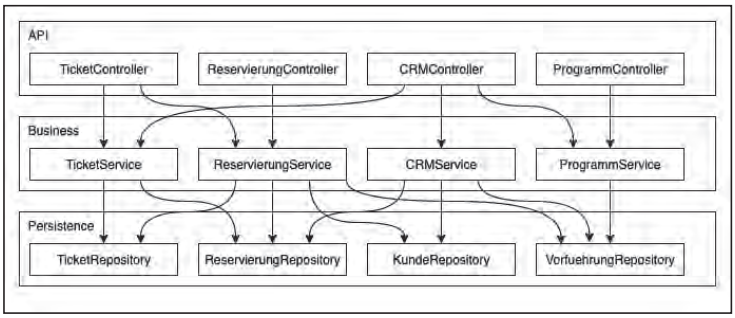


Abb. 2: Reine Schichtenarchitektur

Dieses Paradigma der geringen Kopplung und hohen Kohäsion schließt es bereits aus, Module nach rein technischen Schichten horizontal zu schneiden. Stellen wir uns z. B. ein Modul vor, das als einzige Verantwortlichkeit Persistenz hat. Darin wären Interfaces/Klassen, die jeweils auf „ihre“ Tabelle in der Datenbank zugreifen. Diese Klassen würden sich nie gegenseitig aufrufen, sondern jeweils nur Datenbankzugriffe vornehmen – also geringe Kohäsion innerhalb des Moduls. Die Kopplung mit „Modulen“ anderer Schichten dagegen wäre vermutlich beträchtlich, da jede datenverarbeitende Logik der Anwendung auf dieses eine Persistenzmodul zugreifen müsste. Damit hätten wir gleichzeitig ein weiteres Antipattern geschaffen, den Dependency Magnet – ein Modul, das Abhängigkeiten zu allen anderen Teilen der Anwendung hat. Wenn man seine ganze Anwendung mit solchen horizontalen Schnitten strukturiert, dann hat man am Ende eine reine Schichtenarchitektur ohne Module und mit einem unübersichtlichen und immer größer werdenden Abhängigkeitsknäuel (**Abb. 2**).

Ein besseres Vorgehen zur Bildung von Modulen sind daher vertikale Schnitte nach fachlichen Verantwortlichkeiten (**Abb. 3**). Jedes Modul erhält die Ver-

Was können Microservices, was ein Modulith auch kann?

- Technische Trennung von fachlichen Kontexten
- Kleine, weitgehend unabhängig voneinander entwickelbare Module
- Beherrschbare Komplexität, einfache Erweiterbarkeit
- Klare Abhängigkeiten mit expliziten Schnittstellen, geringes Risiko für unerwartete Nebeneffekte
- Continuous Integration, Continuous Delivery

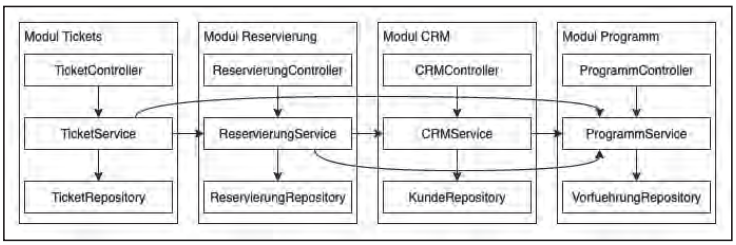


Abb. 3: Strukturierung nach fachlichen Modulen

Abb. 4: Wie bei Hempels unterm Sofa

antwortung für einen klar abgetrennten fachlichen Subkontext und beinhaltet alle Aspekte, die dieser Kontext zum Funktionieren braucht. Es hat die Hoheit über die Daten und die Logik dieser einen Teilmenge der Fachlichkeit und entscheidet, auf welche Weise diese nach außen bereitgestellt werden. Wenn wir uns zum Beispiel das Verwaltungssystem eines Kinos [4] vorstellen, dann haben wir ein Modul für den fachlichen Subkontext „Reservierung“, das die Daten aller Sitzplatzreservierungen hält und die fachliche Logik für das Reservieren und Stornieren von Sitzplätzen implementiert. Ein anderes Modul „Ticket“ hat die Hoheit über den Verkauf von Karten. Es darf keine Reservierungen selbst speichern oder ändern, kann aber dem Reservierungsmodul über eine Schnittstelle mitteilen, dass eine reservierte Karte abgeholt wurde, damit das Reservierungsmodul unter Berücksichtigung aller Reservierungsbusinessregeln die entsprechenden Änderungen an der Reservierung vornehmen kann. Diese Art von Modulschnitt hat noch weitere Vorteile:

- Die Codestruktur bewegt sich nahe an der Fachlichkeit, was für ein einheitlicheres Verständnis zwischen Entwicklern und Fachseite sorgt und die Kommunikation erleichtert.
- Abhängigkeiten zwischen fachlichen Kontexten finden sich 1:1 in den Abhängigkeiten zwischen den Modulen wieder.
- Da neue Features und Erweiterungen meist fachlich getrieben sind, fällt es leicht, die anzupassenden Module zu identifizieren und das Ausmaß der Änderungen darauf zu beschränken. Potenzielle Auswirkungen auf andere fachliche Aspekte werden schnell erkannt, und unerwünschte Nebenwirkungen sind leichter vermeidbar.
- Wenn man es zu einem späteren Zeitpunkt für sinnvoll hält, aus einem fachlichen Subkontext einen eigenen Service zu schneiden, dann ist es einfacher, den Code für den neuen Service zu extrahieren.
- Testbarkeit: Es ist möglich, die fachliche Logik eines Moduls getrennt von anderen Modulen und zusammenhängend durch alle Schichten durchzutesten – von der Schnittstelle bis zur Datenbank.

Um die fachlichen Subkontexte zu identifizieren und gegeneinander abzugrenzen, bietet sich das Konzept der Bounded Contexts [5] aus dem Domain-driven Design an. Ein Bounded Context vereinigt die Sprache, Regeln und Events einer Subdomain und grenzt sie gegen andere Subdomains ab. Methodisch kann man die Definition von Bounded Contexts durch Event Storming oder Context Mapping herauskristallisieren.

Nachdem man seine potenziellen Module identifiziert hat, sollte man sie im Code materialisieren. In einer Java-Anwendung ist der einfachste Weg dafür die Package-Struktur. Jedes Modul besteht aus einem Top-Level Package, in dem man allen Code unterbringt, der zum fachlichen Kontext des Moduls gehört. Im weiteren Verlauf der Artikelserie und in Beispielen wird exemplarisch diese Methode verwendet. Es gibt auch andere Möglichkeiten, Module innerhalb einer Java Codebase abzubilden, worauf ich in einem späteren Abschnitt noch eingehen werde.

Modulinterne Struktur

Wenn wir nun ein Modul definiert, ein Package dafür erstellt und allen Code, der zum Subkontext gehört, hineingeworfen haben, dann kann das am Beispiel unseres Reservierungsmoduls erst einmal so aussehen, wie in **Abbildung 4** gezeigt. Wir sehen eine Menge Code, der sich anscheinend um fachliche Logik rund um die Reservierung kümmert. So weit, so gut. Leider ist es aber noch etwas unübersichtlich. Die Struktur ist undurchsichtig, die Namensgebung inkonsistent und auch die Schnittstelle nach außen ist nicht klar definiert, weil alle Klassen und Methoden den *public*-Modifier haben und damit von außen benutzbar sind. Um den Code modulintern besser zu strukturieren, gibt es unterschiedliche Patterns mit jeweils eigenen Vor- und Nachteilen:

Single Package Modul (Abb. 5): Bei diesem Ansatz verzichten wir auf weitere Unterteilungen innerhalb des

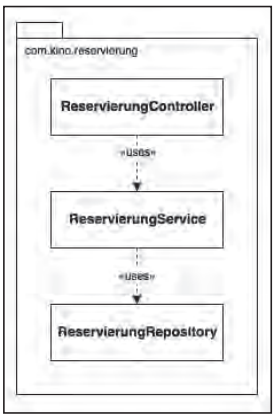


Abb. 5: Die Klassen halten sich an konsistente Namenskonventionen …

… und durch saubere Access Modifier können andere Module nur auf die definierten Schnittstellen des Moduls zugreifen (grünes Schloss)

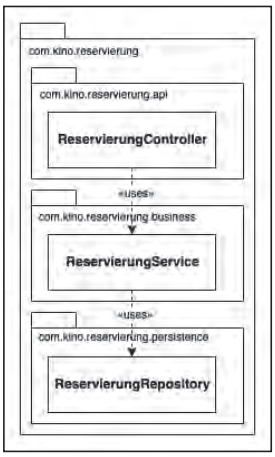
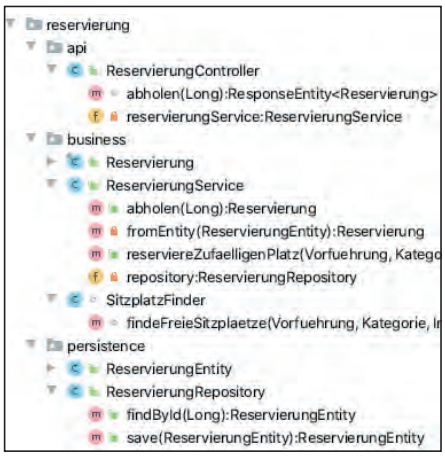


Abb. 6: Übersichtliche Strukturierung …



… auf Kosten der Kapselung interner Logik

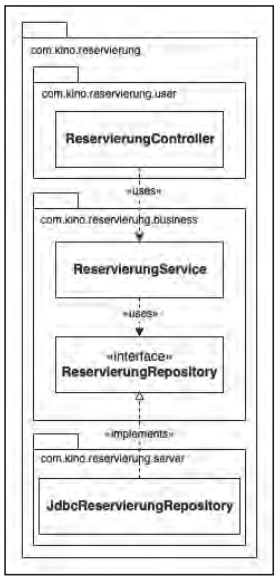
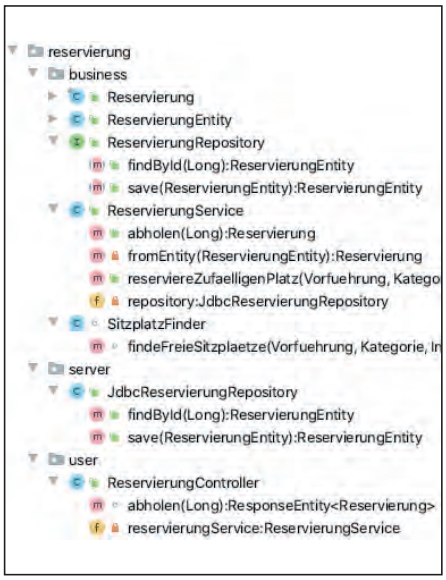


Abb. 7: Die Businesslogik ist gekapselt …



… aber modulinterne Methoden müssen als public deklariert werden

Modul-Packages. Das kann bei großen Modulen der Übersichtlichkeit schaden. Dem kann man durch klare Namenskonventionen, die die Zuständigkeiten der einzelnen Klassen verdeutlichen, etwas entgegenwirken. Ein Riesenvorteil ist, dass hier die Access Modifier in Java voll genutzt werden können. Alle Klassen im Modul sind *package private*, von außen nicht sichtbar und stellen nur *package private*-Methoden bereit. Klasseninterne Methoden sind natürlich *private*. Nur der Business-Service, der die Fachlichkeit des Moduls nach außen bereitstellt, ist *public* und exponiert *public*-Methoden. Auch die zugehörigen Businessobjekte, die nach außen gereicht werden, sind *public*. So wird eine klar definierte Schnittstelle bereitgestellt, die von Clients des Moduls nicht umgangen werden kann..

Slices before Layers (Abb. 6): Dieser Ansatz geht davon aus, dass zuerst vertikale Slices gebildet werden (das sind bereits die Modulschnitte) und diese Slices danach in horizontale, technische Schichten weiter unterteilt werden. Diese Unterteilung findet innerhalb des Moduls durch Unterpackages statt, üblicherweise Persistence, Business, API etc. Dadurch werden die Schichten innerhalb eines Moduls leichter erkennbar und unidirektionale Zugriffe zwischen den Schichten von oben nach unten leichter kontrollierbar. Das dient vor allem der Organisation des Codes und verbessert die Übersichtlichkeit. Allerdings geht diese Aufteilung zu Lasten der Kapselung, denn das Modul muss interne Klassen und Methoden als *public* exponieren, da sonst die Packages der Schichten nicht aufeinander zugreifen können. Dadurch ist es schwieriger, eine klar definierte Schnittstelle bereitzustellen.

Hexagonal Architecture (Abb. 7): Diese auch Ports & Adapters [6] genannte Architekturform ist eigentlich für die Strukturierung einer ganzen Anwendung gedacht, lässt sich aber auch auf den Code innerhalb eines Moduls übertragen. Sie basiert auf dem Prinzip, dass die Domain-Logik in einer zentralen Businesskomponente gekapselt ist und Interaktionen mit der Außenwelt außerhalb davon liegen. Die Außenwelt sind in diesem Fall die User Side (APIs für Clients, GUI etc.) und die Server Side (Datenbank, Dateisystem, andere Server). Dabei hängt immer die äußere Interaktion von der zentralen Businesskomponente ab und nie umgekehrt. Im Fall eines Modulithen können sowohl User Side als auch Server Side Interaktionen mit anderen Modulen innerhalb desselben Modulithen beinhalten.

Der Vorteil ist, dass alle Businessregeln – auch Zugriffe auf Daten und die Art, wie Daten bereitgestellt und manipuliert werden – innerhalb der zentralen Businesskomponente definiert und gekapselt sind. Wie die User Side und Server Side diese Regeln implementieren, ist davon entkoppelt, sie können aber nicht von ihnen abweichen. Dadurch kann man sich bei der Entwicklung gezielt auf einen der Aspekte Businesslogik, Clientinteraktion und Serverinteraktion fokussieren und diese auch entkoppelt voneinander testen.

Eine potenzielle Schwäche ist auch hier: Die Schnittstellen, die die zentrale Businesskomponente innerhalb des Moduls der User Side und Server Side bereitstellt, müssen *public* sein und sind dadurch auch für andere Module sichtbar, obwohl sie nicht unbedingt dafür gedacht sind.

UI Component (Abb. 8): Bei diesem Ansatz wird die dem Client zugewandte Schicht (GUI oder API) zu einer eigenen großen, modulübergreifenden Komponente mit eigenem Top-Level Package. Sie benutzt das API aller Module, um dem Client eine einheitliche Repräsentation der Anwendung bereitzustellen. Die Module bestehen ansonsten wie im Single-Package-Ansatz aus jeweils einem einzelnen Package, das allen Code enthält, der zum Kontext des Moduls gehört, mit Ausnahme des Client API. Jedes Modul stellt dabei ein public API bereit, das idealerweise sowohl von der UI Component als auch von anderen Modulen benutzt werden kann.

Vorteile davon sind, dass die Anwendung dem Client gegenüber wie aus einem Guss präsentiert werden kann und gleichzeitig die Java Access Modifier zur Kapselung

der Module verwendet werden können. Jedes Modul bestimmt vollkommen selbst, wie es verwendet wird, indem es außer dem bewusst bereitgestellten API alles als *package private* deklariert.

Dass die Clientschicht nicht in dieser Kapselung enthalten ist, kann in manchen Szenarien als Nachteil gesehen werden.

Welche der Möglichkeiten die beste ist, ist keine exakte Wissenschaft. In der Entwicklercommunity gibt es für jede der Varianten Befürworter und gute Argumente (beispielsweise [7]). Letztendlich entscheiden die Rahmenbedingungen darüber, welche Vorteile einem Projekt am meisten helfen und auf welche Aspekte hin man seinen Code optimieren möchte. Ist das Ziel ein Serviceschnitt? Dann ist ein Ansatz zu wählen, bei dem man einfach ein Modul, so wie es ist, ausschneiden und in ein frisches Projekt einfügen kann, wie z. B. der Single-Package-Ansatz.

Arbeitet ein großes Team mit unterschiedlichen Erfahrungslevels an dem Projekt? Dann ist der UI-Component-Ansatz vermutlich der beste Weg, da er die beste Kapselung der Module durch Access Modifier erreicht, sodass es schwerer wird, aus Versehen an der Architektur und den Businessregeln vorbei zu entwickeln. Dadurch hätte bestimmt schon der eine oder andere Big Ball of Mud vermieden werden können. Möchte man durch die Strukturierung des Codes in erster Linie Übersichtlichkeit und Codeorganisation und erst in zweiter Linie Kapselung erreichen? Dann bietet sich die Strukturierung nach Slices before Layers an.

Enorm wichtig ist, dass man sich für eine der Varianten entscheidet und diese dann im gesamten Modulithen einheitlich verwendet. Die unterschiedlichen Varianten spielen ihre Stärken erst dann aus, wenn man sich darauf verlassen kann, dass sie für das gesamte Projekt gelten. Springt man stattdessen zwischen den Varianten, holt man sich alle deren Nachteile in den Code, ohne wirklich von ihren Vorteilen profitieren zu können. Zusätzlich stiftet man eine Menge Verwirrung für jeden, der sich an den Code heranwagt.

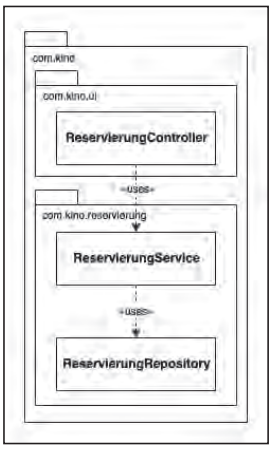
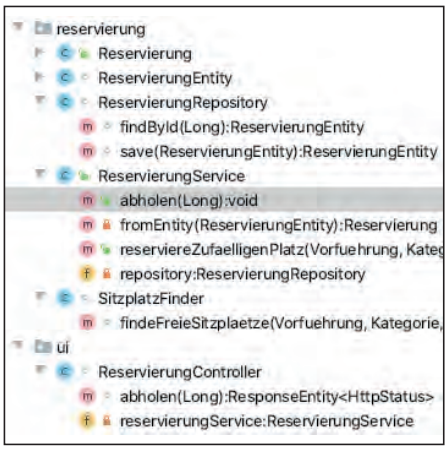


Abb. 8: Die UI-Komponente kann die gleichen Schnittstellen des Moduls benutzen wie andere Module



Die Kapselung bleibt erhalten

Technische Querschnittsmodule

Ein Antipattern, das in fast jedem größeren Projekt zu finden ist, ist das Util-Package. Oft wird es auch „helper“, „common“ oder „base“ genannt, oder es existiert nicht einmal offiziell, da sein Inhalt stattdessen im Basis-Package der Anwendung ausgebreitet ist. Darin befindet sich dann Code, der von mehreren Teilen der Anwendung verwendet wird und deshalb nirgendwo anders zuzuordnen war. Und technische Utilities, die zwar nützlich sind, aber nichts miteinander zu tun haben. Und dann noch technische Querschnittsthemen, die alle Module betreffen und deshalb einfach in das „Package für alles“ mit reingepackt wurden. Und alles andere, über das man halt gerade nicht nachdenken wollte. Ein solches Package erfüllt nicht die Eigenschaften eines Moduls und sollte in einem Modulithen vermieden werden:

- Es hat keine klare Verantwortlichkeit, sondern mehrere unzusammenhängende Zuständigkeiten, was der Name bereits widerspiegelt.
- Es hat eine niedrige Kohäsion, da die verschiedenen Aspekte meist nichts miteinander zu tun haben, sodass es nicht sinnvoll ist, sie zusammen zu gruppieren.
- Es hat eine hohe Kopplung, da jeder Aspekt von irgendeinem Modul gebraucht wird und das Util-Package manchmal sogar zusätzlich alle anderen Teile der Anwendung kennen muss. Oft schließt es damit einen Abhängigkeitskreis über alle Module und wird so zum schlimmsten Dependency Magnet.

Stattdessen sollte man technische und fachliche Querschnittsthemen in einem Modulithen ähnlich behandeln wie fachliche Kontexte. Nehmen wir als Beispiel das Verschicken von E-Mails. Wenn mehrere Module E-Mails versenden müssen, um ihre Fachlichkeit zu erfüllen, dann ist es gut, wenn es ein technisches Modul gibt, das sich um den technischen Aspekt „E-Mails versenden“ kümmert und um nichts anderes. Dieses bietet dann eine von allen Modulen verwendbare Schnittstelle an, die unabhängig von deren Fachlichkeit funktioniert. Allgemeine Logik wie die Komposition der Mail oder Metainformationen der Mail ist in dem E-Mail-Modul gekapselt und es ist das einzige Modul, das die externe Abhängigkeit zum E-Mail-Server kennt. Dabei kennt das E-Mail-Modul keine fachlichen Details wie den Inhalt der Mails. Diese bleiben in der Hoheit der fachlichen Module, die die E-Mails verschicken möchten. Im Beispiel unseres Kinos könnte das Reservierungsmodul nach einer erfolgreichen Reservierung das E-Mail-Modul aufrufen, um dem Kunden eine Mail mit der Reservierungsnummer als Inhalt zu senden, während das Kundenmodul über dieselbe Schnittstelle einen Newsletter mit dem aktuellen Programm verschicken könnte.

Genauso erhalten andere Querschnittsthemen, wie z. B. Securityaspekte, Auditlog, Scheduling oder Reporting ihre eigene Kapselung durch ein eigenes Modul. Als Ergebnis entsteht eine schöne Landschaft aus

fachlichen und technischen Modulen, die ihre Zuständigkeiten sicher kapseln und deren Abhängigkeiten untereinander leichter beherrschbar sind. Das Util-Package ist verschwunden oder zumindest so geschrumpft, dass es nur noch Klassen enthält, die die Bezeichnung „util" auch verdient haben, wie z. B. ein *DateUtil* mit statischen Methoden, die maßgeschneiderte Datumsoperationen bereitstellen.

Toolunterstützte Modulbildung

Bisher ging der Artikel von der einfachsten und gebräuchlichsten Modulbildungsmethode aus, nämlich der Abbildung von Modulen durch Top-Level Packages im Java-Code. Alternativ (Kasten: „Modularisierungstools im Java-Ökosystem") bieten auch Build-Tools wie Maven [8] und Gradle Features [9] wie Multi-Module Builds, mit denen man zumindest Code in Module aufteilen und zyklenfreie Abhängigkeiten zwischen den Modulen sicherstellen kann [10]. Es gibt aber auch Technologien wie OSGi [11] oder Jigsaw [12], das native Modulsystem der Java-Plattform, die sich als ausgewachsene Modularisierungslösungen verstehen und versuchen, das Problem der Modulbildung mitsamt Abhängigkeiten und Schnittstellen expliziter zu lösen.

Durch den Einsatz eines solchen Werkzeugs gewinnt man explizit definierte Module mit explizit definierten öffentlichen Schnittstellen. Das macht es viel einfacher, gesetzte Regeln von Kapselung, Interaktion und Abhängigkeiten der Module einzuhalten. Man muss sich nicht mehr allein auf die Java Access Modifier und Packagestruktur verlassen und ist innerhalb der Module freier mit deren Einsatz und der Codeorganisation durch Subpackages.

Der Preis dafür ist allerdings hoch. Man fügt seinem ohnehin schon komplexen Monolithen mit der Modularisierungsschicht eine weitere Abstraktionsebene und Komplexitätsdimension hinzu, die jeder Entwickler im Team kennen und beherrschen muss Jigsaw [14]. Oft tauchen dadurch auch neue Probleme auf, wie beispielsweise erschwertes Zusammenspiel mit externen Abhängigkeiten, Versions- und Namenskonflikte zwischen Modulen, Inkompatibilitäten zu alten Modulen oder Sprachversionen etc. Das kann zu Frustmomenten und unschönen Workarounds führen, die im schlimmsten Fall mehr schaden können als das Modulsystem genutzt hat.

Die Entscheidung zur Verwendung eines solchen Modularisierungstools muss sehr bewusst getroffen werden. Ich empfehle den Einsatz nur, wenn man damit ein konkretes Problem lösen kann, wie z. B. die Trennung der Module als Deployment-Einheiten oder wenn das Team auf herkömmliche Weise die Modularisierung nicht in den Griff bekommt. Proaktiv auf ein Modulsystem zu setzen wäre ein Fall von YAGNI: „You ain't gonna need it."

Modularisierungstools im Java-Ökosystem

- OSGi [11]
- Java Platform Module System (Jigsaw) [12]
- JBoss Modules [13]
- Maven Multi-Module Projekte [8]
- Gradle Multi-Project Builds [9]

Fazit

Wir haben festgestellt, dass Modulithen eine lohnende Form der Softwarearchitektur sein können. Geschickt geschnittene Module sorgen für Struktur und Übersichtlichkeit in einer großen Codebase, womit sie beherrschbar und testbar wird. Das Innenleben der Module zu gestalten, ist eine Wissenschaft für sich, aber es gibt genügend Möglichkeiten zur Auswahl.

Aber wie kommunizieren Module? Wie managt man die Abhängigkeiten vieler Module, ohne dass sie sich zu einem Knäuel verheddern? Die Antworten auf diese und weitere Fragen wird der nächste Teil dieser Serie geben.

Arnold Franke wirkt als Entwickler und Berater für die Kunden der synyx GmbH & Co KG in Karlsruhe. Dabei baut er als Komplexitätsvermeider und Software Craftsman nachhaltige, pragmatische Lösungen mit sauberem Code.

@indyarni **https://blog.synyx.de**

Links & Literatur

[1] https://martinfowler.com/articles/microservices.html

[2] https://www.thoughtworks.com/insights/articles/demystifying-conways-law

[3] Fowler, Martin: „Refactoring", Improving the Design of Existing Code, Addison Wesley, 2018

[4] https://github.com/indyarni/modulithpatterns

[5] Evans, Eric: „Domain Driven Design", Tackling Complexity in the Heart of Software, Addison Wesley Longman, 2003.

[6] https://alistair.cockburn.us/hexagonal-architecture/

[7] http://www.codingthearchitecture.com/2016/04/25/layers_hexagons_features_and_components.html

[8] https://books.sonatype.com/mvnex-book/reference/multimodule.html

[9] https://guides.gradle.org/creating-multi-project-builds/

[10] https://www.youtube.com/watch?v=bVaiTPYIHFE

[11] https://www.osgi.org

[12] https://www.oracle.com/corporate/features/understanding-java-9-modules.html

[13] https://github.com/jboss-modules/jboss-modules

[14] https://youtu.be/VSiETATcoVw