JAVA Mag

Java™magazin

Java | Architektur | Software-Innovation

Git me baby, one more time

Ausgabe 2.2021

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

4 194586 709801 02

www.javamagazin.de

Architekturpatterns in Modulithen – Teil 3

Die Bude sauber halten

Puh, endlich geschafft. Die Artikelserie geht dem Ende zu. Diese Menge Code zu einem anständigen Modulithen zu formen, war ganz schön anstrengend. Zum Glück ist er jetzt fertig, alle Arbeit ist getan! Wie? Weiterentwicklung? Wartung und Betrieb? Neue Anforderungen? Das Team skalieren? Technische Schulden? Aber das Ding ist doch ganz neu! Warum müssen wir da schon wieder ran? Tja, machste nix. Oder doch?

von Arnold Franke

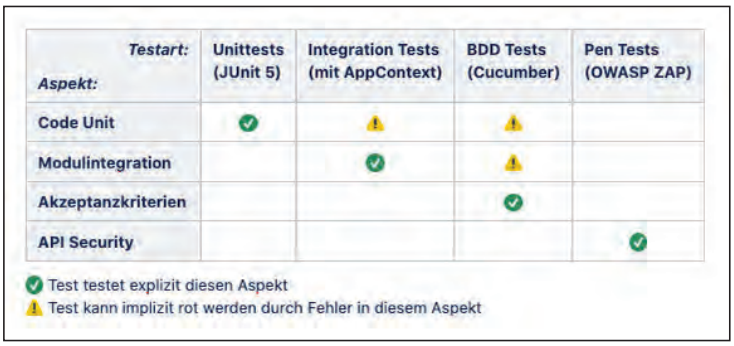
Klar machen wir da was! Genau wie jede andere Software bleibt ein Modulith nur dann am Leben, wenn kontinuierlich weiter an ihm gearbeitet wird. Wenn man nicht ohnehin schon ständig neue Features und geänderte Anforderungen umsetzt, dann sind Sicherheitsupdates, abgekündigte Versionen, sich ändernde Abhängigkeiten, neue Tools und Erkenntnisse genug Gründe für nicht abreißende Weiterentwicklung. In einer großen, modulithischen Codebase gibt es dafür eine Menge Herausforderungen. Das können sein: großflächige Refactorings, häufiges Hinterfragen der System- und Anwendungsarchitektur und das Erreichen einer hohen Testabdeckung aller relevanten Aspekte, um dabei nichts kaputt zu machen. Wenn man das alles im Griff hat, dann spricht auch nichts dagegen, den Modulithen ganz Microservice-like mehrmals am Tag über die Continuous Delivery Pipeline auf Prod zu pushen.

Das Testuniversum

Je größer ein Modulith ist, desto mehr verschiedene Aspekte sind für die reibungslose Funktion entscheidend. Der Code in den Modulen muss tun was er soll, außerdem sollen die Module zusammengenommen das

Artikelserie

Teil 1: Ordnung ins Chaos bringen
Teil 2: Wer mit wem reden darf
Teil 3: Die Bude sauber halten



Testart: / Aspekt:	Unittests (JUnit 5)	Integration Tests (mit AppContext)	BDD Tests (Cucumber)	Pen Tests (OWASP ZAP)
Code Unit	✓	⚠	⚠	
Modulintegration		✓	⚠	
Akzeptanzkriterien			✓	
API Security				✓

✓ Test testet explizit diesen Aspekt
⚠ Test kann implizit rot werden durch Fehler in diesem Aspekt

Abb. 1: Die Testmatrix visualisiert, welcher Test was testet

erwartete Verhalten an den Tag legen. Die Integration mit externen Abhängigkeiten und internen beweglichen Teilen wie Datenbanken, Caches und Events muss stabil laufen. Es gibt nichtfunktionale Anforderungen wie Performance, Resilience, Security. Fachliche Akzeptanzkriterien müssen eingehalten werden. Wie soll man all das testen und dabei nicht verrückt werden?

Es ist keine schlechte Idee, sich erst einmal an der klassischen Testpyramide [1] zu orientieren. Eine möglichst komplette Unit-Test-Abdeckung möchte man wahrscheinlich in den meisten Projekten haben. Aber braucht man Lasttests, wenn man nur 1000 Requests pro Tag beantwortet? Wie wichtig ist das automatisierte Sicherstellen von Performance, wenn kein echter User auf Antwort wartet? Welche Integrationstests braucht das Projekt unbedingt, welche sind vernachlässigbar? Muss man jeden fachlichen Edge Case abdecken oder reicht der Happy Path? Solche Fragen sollte man sich stellen und sich dann bewusst entscheiden, welchen Aspekt man in welcher Intensität automatisiert testen will. Dabei wird man den einen oder anderen Kompromiss zwischen kompletter Abdeckung aller Aspekte und dem dafür notwendigen Aufwand eingehen müssen.

Um ein einheitliches Bild in den Köpfen des Teams zu etablieren, eignet sich eine Visualisierung aller zu testenden Aspekte und der zugehörigen Testarten wie die Tabelle in **Abbildung 1** zeigt.

Dort sieht man auf den ersten Blick anhand der grünen Haken, welche Art von Test in welcher Technologie für das Testen welches Aspekts zuständig ist. Die gelben Symbole visualisieren unvermeidbare Überschneidungen. So kann es beispielsweise passieren, dass durch eine fehlerhafte Modulintegration auch ein fachlicher Akzeptanztest auf die Bretter geht, weil er diese Integration durchläuft, obwohl er sie gar nicht explizit testen will. Die API-Security-Tests dagegen werden dadurch nicht behelligt.

In einem großen Modulithen hat so eine Matrix für gewöhnlich noch deutlich mehr Zeilen und Spalten als in **Abbildung 1**. Man kann so eine Definition des Testuniversums in einem Teamworkshop als Ist-Zustand und Soll-Zustand erarbeiten. Am Ende hat jeder ein einheitliches Bild im Kopf und weiß, wo die Reise hingeht. Testentwicklung läuft koordinierter ab, während das Team Sicherheit gewinnt, denn fehlende Tests gehen einem jetzt nicht mehr so schnell durch die Lappen, und man weiß genau, wo man hin greifen muss, wenn ein Test rot wird.

Wie wichtig ist die Testabdeckung?

Ergänzt wird das Konzept durch Meaningful Test Coverage [2]. Das bedeutet zunächst, dass man nicht nur sinnlose Tests schreibt, die einfach jede Zeile Code durchlaufen, um eine hohe Coverage zu erzielen. Jeder Test sollte einen sinnvollen Aspekt testen und entsprechend spezifische Assertions durchführen. Außerdem bedeutet es, dass man sich bewusst entscheidet, manchen Code **nicht** zu testen, wie z. B. generierten Code, Testcode, Konfigurationscode, und ihn dann auch aus der Coverage-Analyse auszuschließen. Auf fachlicher Ebene kann das bedeuten, dass man vielleicht nicht jeden fachlichen Edge Case automatisiert testen will, sondern nur die essenziellen Use Cases. Bei solchen Entscheidungen sollten die entsprechenden Stakeholder ein Wörtchen mitzureden haben. Am Ende steht das hehre Ziel, möglichst 100 Prozent der Dinge, die man tatsächlich testen möchte, durch sinnvolle Tests abzudecken. Das gilt für die Zeilen Code bei Unit-Tests, für Integrationspunkte bei Integrationstests, fachliche Aspekte bei Akzeptanztests und entsprechend auch für alle anderen Arten von Tests.

Isolation von Tests

Mit einem Test nur exakt den Aspekt anzusprechen, den man damit auch testen will, ist oft gar nicht so einfach. Jedes getestete Stück Code verwendet andere Klassen oder Abhängigkeiten, die man eigentlich nicht mittesten will. Wenn man für den Integrationstest einer Schnittstelle die ganze Anwendung hochfährt, braucht man auch alle Abhängigkeiten wie Datenbank, Filesystem, Messagingsysteme, die mit der Schnittstelle gar nichts zu tun haben. In der komplexen Codebase eines Modulithen ist das so viel, dass Laufzeit, Ressourcenverbrauch und Aussagekraft der Tests stark darunter leiden. Daher ist die Isolation des zu testenden Aspekts beim Testaufbau eine Kunst für sich. Im Modulithen trifft man auf unterschiedlichste Isolationsszenarien. Daher ist es sinnvoll, dafür einen Werkzeugkasten mit den richtigen Isolationsmethoden parat zu haben.

Auf Unit-Test-Ebene gibt es für Java eine Menge an Mocking Libraries wie Mockito [3] und Powermock [4], die es leicht machen, eine Code Unit zu isolieren, indem man Mocks für alle Abhängigkeiten konfiguriert. Im Spring Framework gibt es viele Möglichkeiten, für integrative Tests nur genau die Teile der Anwendung hochzufahren, die man braucht, zum Beispiel die Spring Boot Test Slices [5] oder eigene Context-Konfigurationen. Auch sehr nützlich ist das realitätsnahe Nachstellen von Requests auf eigene Web-Controller mit Spring MockMVC [6].

Um externe Systeme in Tests außen vor zu lassen, kann man in einem eigenen Testprofil die Implementierungen der Interfaces zu den externen Systemen mit

Die Boyscout Rule ist die wichtigste Regel: jedes Stück Code, das man anfasst, sauberer verlassen, als es war – dazu gehört, das geplante Refactoring an dieser Stelle durchzuführen.

Mocks ersetzen und hat damit seine Anwendung nach außen isoliert. Eine (oft bessere) Alternative hierzu ist es, stattdessen die externe Abhängigkeit selbst durch einen Testmechanismus zu ersetzen und damit die Integration zur Abhängigkeit mitzutesten. Zum Beispiel kann man mit Wiremock [7] mit geringem Aufwand externe Webserver simulieren und genau konfigurieren, wie sie sich im Test verhalten sollen. Andere externe Abhängigkeiten wie Message-Systeme, Datenbanken, Verzeichnisdienste können mit den auf Docker aufbauenden Testcontainers [8] sehr schnell während des Tests hoch- und runtergefahren werden (Listing 1). Im Test wird ein Container mit einer echten MySQL gestartet. *@DataJpaTest* fährt den benötigten Slice des Spring Contexts hoch.

Refactoring im großen Stil

Refactorings in einem Monolithen sind manchmal furchteinflößend. Wenn man einfach drauflos wurschtelt, verzettelt man sich schnell oder kommt in die Situation, Änderungen in hunderten Dateien auf einmal zu mergen. Oder man manövriert sich in aussichtslose Lagen, in denen man die ganze Arbeit wieder zurückrollen kann. Daher sollte man für große Refactorings einen Plan haben. Nach Schema F kann man dabei leider nur selten vorgehen, es gibt aber ein paar hilfreiche Methoden und Arbeitsweisen, die sich grob in „hilfreich für Fleißarbeit“ und „hilfreich für Strukturänderung“ unterteilen lassen.

Refactoring – Fleißarbeiten

Hierzu zählen Refactorings, die an sich nicht komplex sind, aber viele Stellen in der Codebase betreffen und daher nicht mal eben schnell runtergehackt sind. Ein Beispiel: Man will im gesamten Code alle Dependency Injections von Field Injection auf Constructor Injection umbauen, da man das für sinnvoll hält [9]. Oder man will einen Großteil der Integrationstests in Unit-Tests überführen. Das kann in einem großen Monolithen hunderte Klassen betreffen, die man von Hand ändern muss. Die Stellen zu identifizieren, ist in einem Monolithen meist einfacher als im verteilten System, denn man hat ja den gesamten Code in einem Repo. Eine Volltextsuche oder die IDE listen schnell alle Baustellen auf.

Je nach Dringlichkeit und Größe des Refactorings kann man natürlich einen Unglücksraben bestimmen, der u. U. über Wochen alles auf einmal durchrockt und sich danach nochmal genauso lange Urlaub nehmen muss. Oft ist es aber klüger, den Zustand über die Zeit hinweg mit kleinen Änderungen zu verbessern. Dafür ist es erst einmal notwendig, dass jedes Teammitglied den Zielzustand genau kennt und den Grund für das Refactoring versteht. Das erreicht man am einfachsten, wenn man die Änderung an einem echten Beispiel gemeinsam durchspielt. Danach ist die wichtigste Regel die Boyscout Rule [10]. Man verlässt jedes Stück Code, das man anfasst, sauberer, als man es angetroffen hat – und dazu gehört auch, das geplante Refactoring an dieser Stelle durchzuführen. So führt sich das Refactoring über die Zeit „von allein“ durch, bis man dann irgendwann den letzten Rest in einem Rutsch wegarbeitet.

Für das Boyscouting und die Beobachtung des Fortschritts ist es manchmal hilfreich, alle zu ändernden Stellen im Code zu markieren, wofür sich in Java zum Beispiel Marker-Annotationen eignen. Man erstellt eine Annotation wie *@ShouldBeAUnitTest* und klatscht sie erst einmal an jede Testklasse, die man umbauen will. Wer über die Annotation stolpert, der weiß dann gleich, was zu tun ist, und man kann von der IDE leicht die verbleibenden Stellen zählen lassen. Codekommentare sind dafür ein unzureichender Ersatz, da sie nicht so einfach referenzierbar und anfällig für Tippfehler sind.

Es kann sich auch lohnen, eine Refactoring-Kennzahl einzuführen, die den kontinuierlichen Fortschritt greifbar macht und immer präsent ist. Zum Beispiel ist es möglich, ein Plug-in für SonarQube zu schreiben, das in jeder statischen Codeanalyse die Anzahl der verbleibenden Field Injections ausweist [11]. Das Gleiche lässt

Listing 1

```
@DataJpaTest
class ReservierungRepositoryTest {
@Autowired
  private ReservierungRepository reservierungRepository;
@Test
  void testMySql() {
    try (MySQLContainer<?> mySqlContainer = new MySQLContainer<>()
        .withDatabaseName("kino")
        .withUsername("foo")
        .withPassword("bar")) {
    mySqlContainer.start();
      assertThat(reservierungRepository.findById(1L)).isNotNull();
    }
  }
}
```

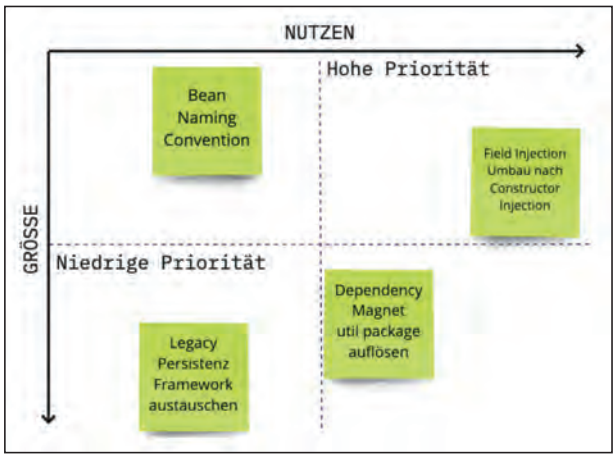


Abb. 2: Das Schulden-Board visualisiert und priorisiert technische Schulden

sich per ArchUnit [12] auch zum Testzeitpunkt prüfen und in eine Datei oder einen beliebigen anderen Store ausgeben. So kann man die Kennzahl nach und nach verbessern, ohne dass deswegen gleich Tests fehlschlagen (Listing 2).

Strukturelle Refactorings

In den ersten beiden Teilen der Serie haben wir viel Zeit damit verbracht, die interne Architektur des Modulithen mit Modulschnitten und bewussten Abhängigkeiten zu modellieren. Um diese hohe Qualität in der Struktur zu halten, muss man sie während der Weiterentwicklung immer wieder hinterfragen. Sollte man hier eine Abstraktion einführen? Verdient dieser Subcontext ein eigenes Modul? Sollte die Abhängigkeit dieser beiden Module nicht umgedreht werden? Dadurch identifizierte strukturelle Refactorings sind häufig größerer Natur. Man geht sie am besten an, indem man sie möglichst in Teilschritte zerlegt, die jeweils einen funktionierenden und deploybaren Zwischenstand haben. Durch häufigeres Mergen kleinerer Änderungen und einen kürzeren Feedbackzyklus kann man das Risiko eines großen Refactorings reduzieren und auftretende Probleme gezielter beheben. Wenn man ein großes Refactoring auf einmal ohne Zwischenschritte durchführt, dann ist auch der Schmerz beim Merge größer, und bei danach entdeckten Bugs ist es schwieriger, die Ursache davon in dem Riesen-Merge zu identifizieren.

Es gibt Tools, mit denen man so ein großes Refactoring im Voraus durchspielen kann. In einem Structure101-Architekturdiagramm [13] kann man Klassen oder Packages per Drag and Drop verschieben und sieht sofort, welche Auswirkungen die Änderung hat. So stellt man oft schon vorher fest, ob ein Refactoring den gewünschten Effekt hat, kann potenzielle Probleme identifizieren und Teilschritte ableiten. Einen gewissen Schutz gegen unbeabsichtigte Verletzungen der Architektur bieten automatisierte Strukturtests, wie z. B. ArchUnit, die Architekturregeln definieren, gegen die der Code bei jedem Testlauf geprüft wird.

Listing 2

```
@Test
void checkRemainingFieldInjections() {
  JavaClasses classes = new ClassFileImporter().importPackages("com.kino");
ArchRule rule = FreezingArchRule.freeze(
            fields()
              .should()
              .notBeAnnotatedWith(Autowired.class));
rule.check(classes);
}
// Führt zum diesem Fileoutput:
Field <com.kino.export.ExportService.kundeService> is annotated with
                                    @Autowired in (ExportService.java:0)
Field <com.kino.export.ExportService.reservierungService> is annotated
                                with @Autowired in (ExportService.java:0)
```

Technische Schulden

Der Beweggrund für Refactorings sind häufig technische Schulden – also technische Mängel, die man in der Vergangenheit bewusst in Kauf genommen hat oder die sich über die Zeit angesammelt haben. Das ist per se nichts Schlimmes, solange man nicht die Kontrolle darüber verliert. Gerade in großen Projekten besteht die Gefahr, dass unbemerkt ein Berg entsteht, der nie wieder abzutragen ist und die Weiterentwicklung auf ewig bremst. Daher ist es wichtig, Auswirkung und Umfang der technischen Schulden immer im Blick zu haben und zu kontrollieren, indem man den Berg durch gezielte Refactorings kontinuierlich verringert.

Für Fans von Post-Its ist das technische Schulden-Board ein guter Weg, den Überblick über den Berg zu behalten (**Abb. 2**). Es handelt sich um ein Priorisierungs-Board mit zwei Achsen für den Mehrwert und die Größe einer technischen Schuld. Jedes Mal, wenn das Team eine technische Schuld identifiziert, wird sie auf dem Board in den beiden Achsen eingeordnet. Diese Visualisierung hilft bei der Einschätzung der Gesamtschuld und bei der Priorisierung der nächsten Refactorings – die kleinsten Schulden mit dem höchsten Mehrwert werden zuerst bearbeitet.

Statische Codeanalyse kann sowohl bei der Erkennung als auch bei der Verfolgung von technischen Schulden helfen. In Tools wie SonarQube [14] erkennt man Hotspots zum Beispiel schnell an Kennzahlen wie der Cyclomatic Complexity [15] und der Duplikationsrate. Auch generelle Gesundheitsmetriken wie die Größe von Klassen und Funktionen können Hinweise auf problematische Stellen geben. Wie oben beschrieben, kann man technische Schuld auch durch eigene Kennzahlen visualisieren, die man in einem SonarQube-Plug-in anzeigt oder von automatisierten Tests ausgeben lässt.

Die Reißleine

Im schlimmsten Fall kann es passieren, dass ein großes Softwareprojekt durch eigene und äußere Restriktionen in einen so maroden Zustand gerät, dass die Wei-

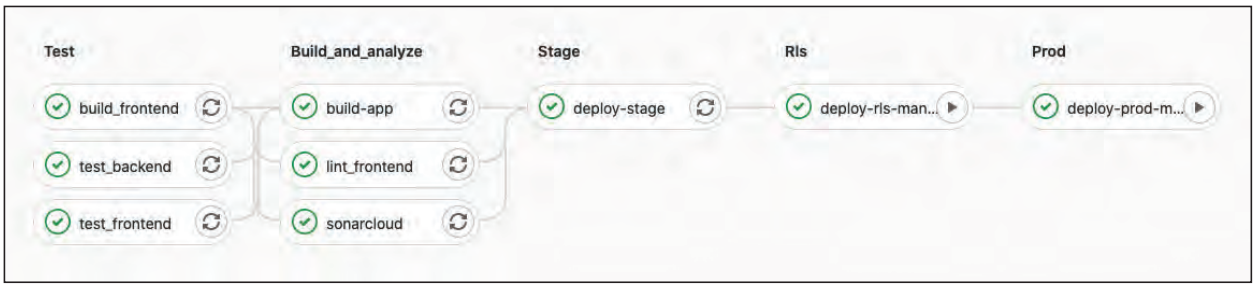


Abb. 3: Eine CI/CD Pipeline dauert beim Modulithen etwas länger, ist aber unverzichtbar

terentwicklung zu langsam und aufwendig wird, um sich zu lohnen, oder das Risiko zu groß wird, dabei etwas kaputtzumachen. Man spricht dann von einem Big Ball of Mud. Bevor man aufgibt und alles wegschmeißt, kann man immer noch die Reißleine ziehen und eine technische Rettungsphase starten. Man identifiziert die größten Probleme, die eine Weiterentwicklung verhindern, und fokussiert sich darauf, diese mit gezielten Refactorings zu beseitigen, während man die fachliche Weiterentwicklung stoppt oder zumindest auf das absolut Nötigste reduziert. Ein solcher Kraftakt ist für alle Stakeholder unangenehm, aber oft die bessere Alternative zur Kapitulation. Damit einhergehen muss aber auch ein Kulturwandel, der neue Arbeitsweisen und Mindsets mit sich bringt – sonst wird sich an der Qualität des Ergebnisses auch nichts ändern und am Ende des Rettungsversuchs steht gleich der nächste Big Ball of Mud.

Weiterentwicklung des Modulithen

Bei Software in der Größenordnung eines Modulithen ist der Funktionsumfang für gewöhnlich nie „komplett“. In einer großen fachlichen Domain gibt es immer Raum für neue Ideen und Funktionen, für einfachere Bedienung und mehr Automatisierung. Es gibt viele Voraussetzungen zu erfüllen, um den Entwicklungs-Flow nachhaltig zu sichern.

Continuous Delivery

Die Größe des Modulithen sollte keinen Einfluss darauf haben, wie oft der aktuelle Codestand an den Benutzer ausgeliefert wird. Wenn alle wichtigen Aspekte durch automatische Tests abgedeckt sind, das Entwicklungsteam seinen Code kontinuierlich integriert und sowohl Build-Prozess als auch Deployment automatisiert sind, spricht nichts dagegen, Continuous Delivery zu praktizieren und jedes fertige Feature sofort auf Produktion auszurollen (**Abb. 3**). Natürlich führen lange Compile- und Build-Zeiten, die Menge an Tests und die Größe des Artefakts dazu, dass dieser Prozess länger dauert als beim typischen Microservice. Aber es ist immer noch besser, ihn kontinuierlich anzustoßen als nur alle x Wochen nach festem Releasezyklus. Dadurch verringern sich die Time to Market, das Risiko und die Kosten pro Release. Die Qualität der Software ist immer in auslieferbarem Zustand, und langfristige Planungen werden nicht durch den Releasezyklus eingeschränkt.

Das Festhalten an einem Releasezyklus kann je nach Projektcharakter und Rahmenbedingungen gute Gründe haben, ist aber häufig nur ein Hinweis, dass das Vertrauen in Qualität und Tests der Software noch nicht ausreicht. Zum Glück sind das meist Dinge, die man selbst in der Hand hat und die man aus eigener Kraft in einen Continuous-Delivery-fähigen Zustand bringen kann.

Cognitive Load und Komplexität

Ein Modulith wird optimalerweise von einem Team moderater Größe weiterentwickelt und betreut. Dabei muss das Team die fachliche Domain gut kennen, wissen, wie diese im Code umgesetzt ist, alle eingesetzten Technologien beherrschen, alle Abhängigkeiten kennen, die Architektur im Blick haben, die Betriebsinfrastruktur verstanden haben und so weiter und so fort. Alle Aspekte, die zur Entwicklung des Modulithen gehören, müssen in den Köpfen der Teammitglieder gleichzeitig Platz haben – man nennt das Cognitive Load [16]. Wenn diese zu groß wird und die Köpfe des Entwicklungsteams „voll“ sind, leidet darunter sowohl die bestehende Software als auch jede Weiterentwicklung. Teile des Systems werden vernachlässigt, weil man sich nicht mehr um alles kümmern kann. Die Qualität und die Umsetzungsgeschwindigkeit sinken, weil man nicht mehr alle Zusammenhänge im Kopf haben kann. Es bleibt weniger Zeit für Weiterentwicklung, weil der Erhalt des Status quo so viel Anstrengung bedeutet.

Das ist einer der vielen Gründe, warum es so unglaublich wichtig ist, das KISS-Prinzip (Keep it simple, stupid) [17] auf allen Ebenen einzuhalten und stets den einfachsten möglichen Weg zu wählen, um ein Problem zu lösen. Es ist eine der wichtigsten Voraussetzungen für nachhaltige Software, dass man es schafft, die Komplexität niedrig zu halten, während man weiterentwickelt und verbessert. Dementsprechend ist die Vermeidung von Komplexität auch einer der wichtigsten Skills eines Entwicklers.

Komplexität skalieren

Nun ist es leider nahezu unmöglich, steigende Komplexität zu vermeiden, wenn man den Funktionsumfang einer Software ständig erweitert. Klar kann man das Team vergrößern, um die Cognitive Load weiter zu verteilen, aber Teams skalieren nur begrenzt. Ab acht Entwicklern wird es meist kritisch mit dem Kommunikations- und

Koordinations-Overhead [18]. Irgendwann hat man einen Punkt erreicht, an dem man die Komplexität der Software und die Größe des Teams nicht mehr erhöhen kann.

Will man dennoch mehr skalieren und die Software erweitern, dann bleibt nur der Divide-and-Conquer-Ansatz [19]. Man löst einen Teil des Modulithen heraus und macht daraus einen eigenen Service mit eigener Deployment-Einheit. Aus dem großen Team werden zwei Teams, die sich die Cognitive Load untereinander aufteilen und dann wieder unabhängig voneinander skalieren können.

Eine Alternative dazu ist es, den Teamfokus nach und nach von Weiterentwicklung auf Wartung und Betrieb zu verschieben. Das ermöglicht es, bei gleicher Teamgröße und immer langsamer wachsender Komplexität die Software weiter zu warten und aktuell zu halten. Dieser Schritt geht unweigerlich einher mit einer neuen Zusammensetzung der Rollen im Team. Die Pioniere und Erfinder machen Platz für Städtebauer und Umsetzer [20], [21]. Das ist keine negative Entwicklung, wenn man sie bewusst in Kauf nimmt und nicht von ihr überrascht wird.

Einen neuen Service extrahieren

Das Aufteilen eines Modulithen in zwei oder mehr Services ist aus vielen Gründen nicht trivial. Es gibt neue Herausforderungen in der Infrastruktur, im Betrieb, in der Organisationsstruktur, bei Themen wie Performance, Resilience und Monitoring. Auf all das möchte ich hier nicht eingehen und lieber auf Kollegen verweisen, die das schon sehr gut abgedeckt haben [22].

Wenn man die reine Codesicht und Anwendungsarchitektur betrachtet, sollte es aber recht einfach sein, einen Service aus einem gut strukturierten Modulithen herauszuschneiden. Man entscheidet sich für ein Modul, das einen eigenen, gut gekapselten Subcontext der Domain enthält, und schiebt es komplett in ein neues Repo. Abhängigkeiten zu anderen Modulen werden durch Netzwerk-Calls ersetzt. Im ersten Wurf werden aus Methodenaufrufen meist HTTP Calls und aus Application Events werden Events auf einem externen Event Bus. Dabei ist auch bei den neu entstandenen Services darauf zu achten, dass es keine zyklischen Abhängigkeiten zwischen den Services gibt, sonst baut man wieder einen Deployment-Monolithen.

Leider gibt es bei so einer Trennung auch großes Potenzial für Duplikationen, wenn technische Querschnittsthemen oder Datenstrukturen in beiden Services gebraucht werden. Der oben genannte Overhead für „Dinge außenrum“ kommt noch dazu. Letztendlich ist es immer ein Trade-off, den man damit eingeht, und für den man sich sehr bewusst unter Abwägung aller Vor- und Nachteile entscheiden muss.

That’s all Folks

Jetzt stehen wir hier mit vollem Kopf, die Synapsen überwältigt mit Patterns, Methoden, Tools, Best Practices. Wenn man das jetzt alles so macht wie es die Artikelserie sagt, und genauso anwendet wie beschrieben, dann kommt doch unglaublich tolle Software dabei raus, oder? Ein Modulith der Glückseligkeit und viel besser als mit Microservices? Vielleicht. Vielleicht auch nicht. Gute Software entsteht nicht ohne gesunden Menschenverstand, ohne ständiges Hinterfragen und ohne über den Tellerrand zu gucken. Die Methoden aus den Artikeln haben sich in Problemen aus der Praxis bewährt, aber man soll ja nicht Lösungen für irgendein fremdes Problem finden, sondern für sein eigenes. Darum hoffe ich, dass ich die Werkzeugkiste der Leser ein bisschen erweitern und den Sinn für saubere, modulare Architektur schärfen konnte, ohne den Horizont dabei zu sehr einzuschränken. Ich wünsche viel Spaß beim Refactoring!

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/bliki/TestPyramid.html

[2] https://synyx.de/blog/code-coverage-with-significance/

[3] https://site.mockito.org

[4] https://github.com/powermock/powermock

[5] https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-test-auto-configuration.html

[6] https://spring.io/guides/gs/testing-web/

[7] http://wiremock.org

[8] https://www.testcontainers.org

[9] http://olivergierke.de/2013/11/why-field-injection-is-evil/

[10] https://www.oreilly.com/library/view/97-things-every/9780596809515/ch08.html

[11] https://docs.sonarqube.org/latest/extend/developing-plugin/

[12] https://www.archunit.org

[13] https://structure101.com

[14] https://www.sonarqube.org

[15] https://blog.ndepend.com/understanding-cyclomatic-complexity/

[16] https://techbeacon.com/app-dev-testing/forget-monoliths-vs-microservices-cognitive-load-what-matters

[17] https://t2informatik.de/wissen-kompakt/kiss-prinzip/

[18] https://www.toptal.com/product-managers/agile/scrum-team-size

[19] https://effectivesoftwaredesign.com/2011/06/06/divide-and-conquer-coping-with-complexity/

[20] http://agilebusinessmanifesto.com/agilebusiness/a-structure-for-continuous-innovation-pioneers-settlers-town-planners/

[21] https://de.wikipedia.org/wiki/Teamrolle#Teamrollen_nach_Belbin

[22] https://synyx.de/events/jug-hessen-vom-monolithen-zu-microservices-ein-erfahrungsbericht/