Design for Change: Effiziente Softwareentwicklung

Ingmar Kellner, hello2morrow GmbH

In einer global vernetzten Welt ist die Konkurrenz immens. Dies gilt besonders für den Bereich Softwareentwicklung, wo es meistens keine Rolle spielt, ob der Programmierer in Deutschland, Estland, Indien oder den USA sitzt. Um auf Dauer mit einem Projekt erfolgreich zu sein, wird es daher immer wichtiger, Funktionalität effizient umzusetzen. In diesem Artikel werde ich den Einfluss von Kopplung auf die effiziente Programmierung herausstellen und darüber hinaus, wie man diese minimiert und automatisiert kontrollieren kann.

Um herauszufinden, wie sich die Softwareentwicklung effizienter gestalten lässt, muss man sich darüber klar werden, wo und wann Probleme und Kosten entstehen: Üblicherweise besteht ein kleiner Teil der Kosten in Software-Projekten aus Hardware, beispielsweise für Laptops und Büroausstattung. Der Großteil liegt im personellen Bereich, das heißt, dominant sind hier die zu zahlenden Arbeitsstunden der Entwickler. Will man also eine effizientere Entwicklung, muss die Arbeitszeit der Entwickler besser genutzt werden.

Mit „agilen" Entwicklungsprozessen werden Probleme gelöst, die „Wasserfall"-artig durchgeführte Projekte mit sich bringen: Lange Analyse- und Design-Phasen mit später Validierung durch Anwender, wodurch oftmals nicht die richtigen Funktionalitäten entwickelt werden, werden ersetzt durch die schnelle Rückkopplung und den sinnvollen Fokus auf das Minimum Viable Product (MVP, zu Deutsch „minimal brauchbares Produkt"). Es werden permanent neue Features entwickelt oder existierende den geänderten Kundenwünschen angepasst. Dabei entwickelt sich auch das Wissen des Entwicklungsteams über die Domäne weiter und diese Erkenntnisse fließen über Refactorings wiederum als Änderungen in die Codebasis ein.

Es haben sich also Methoden etabliert, wie sich die „richtige" Funktionalität entwickeln lässt. Legen wir nun den Fokus darauf, wie besser „richtig" entwickelt werden kann und was notwendig ist, um Änderungen effizienter zu implementieren.

Dabei abstrahieren wir von Technologie und betrachten das Thema unabhängig von existierenden Frameworks wie etwa Spring oder Java EE.

Kosten von Änderungen

Die folgenden Sätze klingen sicher trivial, bilden jedoch die notwendige Begründung für die Wichtigkeit des Themas „Kopplung": Bei der Anpassung von Funktionalität muss man bestehenden Code ändern. In der Regel ist das auch der Fall, wenn neue Features eingebaut werden, denn diese werden in das bestehende System integriert. Je geringer die Wechselwirkungen im Code, desto weniger Code muss angepasst werden.

Die Wechselwirkung zwischen Teilen eines Systems bezeichnet man auch als „Kopplung" oder „Abhängigkeit". Am einfachsten ist diese zu erkennen, wenn die Abhängigkeiten durch Referenzen im Code erfolgen, etwa indem die Klasse die Klasse Customer verwendet, wie in *Abbildung 1* in Form eines UML-Diagramms dargestellt:

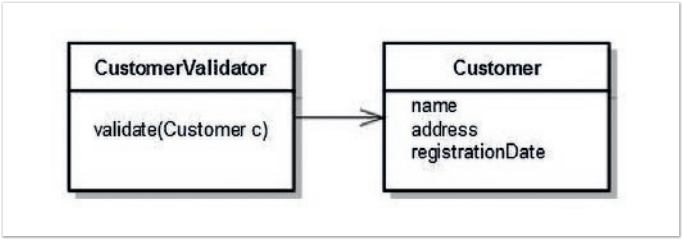

Abbildung 1: Kopplung zwischen Klassen (© Ingmar Kellner)

Ändert sich die Klasse Customer, etwa wenn ein neues Attribut hinzugefügt wird, muss die Klasse `CustomerValidator` ebenfalls angepasst werden, da alle Attribute validiert werden. Je weniger Referenzen auf Customer im Code existieren, desto weniger kann das System durch Änderungen an Customer beeinflusst werden, was zu geringeren Aufwänden und damit geringeren Kosten führt. Der Ansatz, die Abhängigkeiten zu minimieren (etwa, indem Interfaces eingeführt werden für die Umsetzung des „Dependency Inversion Principle"), wird auch als „lose Kopplung" bezeichnet.

Es gilt allerdings zu beachten, dass die oben genannte statische Kopplung nicht die einzige Form der Kopplung ist. Kopplung kann auch über gemeinsam verwendete Datenstrukturen entstehen, beispielsweise über Keys in Maps, zeitlich über ein festgelegtes Protokoll von Methodenaufrufen etc. Explizite Kopplung, die sich über Referenzen im Code nachverfolgen lässt, ist für den Entwickler am einfachsten zu verstehen und lässt sich über Werkzeuge automatisiert überwachen. Aber dazu später mehr.

Erstrebenswert ist also ein Design, in dem Änderungen möglichst lokale Auswirkungen haben. Hat man während der Umsetzung das Gefühl, dass sich die notwendigen Anpassungen wie ein Lauffeu-

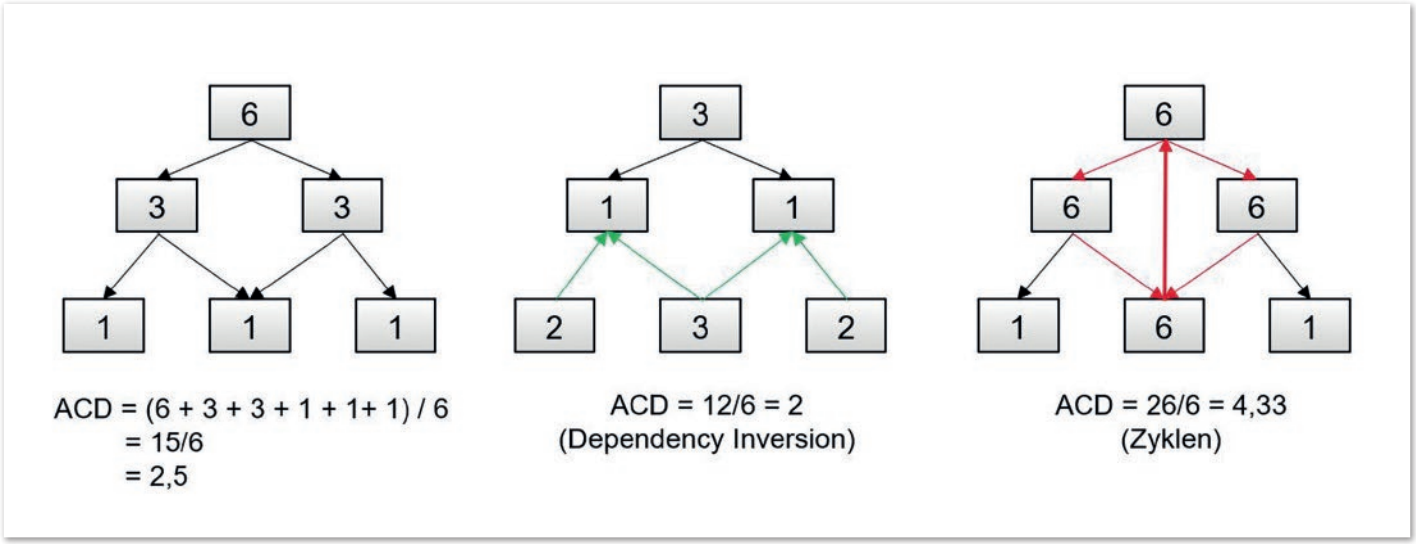

Abbildung 2: „Average Component Dependency"-Berechnung (© Ingmar Kellner)

er auf viele eigentlich unbeteiligte Klassen des Systems auswirken, dann ist das ein Warnzeichen für eine starke Kopplung. Schauen wir uns nun an, wie Kopplung messbar ist, bevor wir uns damit beschäftigen, wie sie sich minimieren lässt.

Messen von Kopplung

Es gibt einige Metriken, um die Kopplung eines gesamten Systems wie auch einzelner Klassen zu bestimmen. Am einfachsten zu verstehen finde ich die von John Lakos beschriebenen Metriken „Depends Upon" und „Average Component Dependency" (ACD) *[1]*. Der Begriff „Komponente" ist hierbei abstrakt zu verstehen und kann ein Modul, ein Package, eine Datei oder eine Klasse sein. „Depends Upon" berechnet sich aus der Anzahl direkter und indirekter Abhängigkeiten (+1 für die Komponente selbst). „Average Component Dependency" ist nun der Durchschnitt der „Depends Upon"-Werte. In *Abbildung 2* wird die Berechnung am Beispiel von wenigen Komponenten mit unterschiedlichen Abhängigkeiten deutlich. Der positive Effekt des „Dependency Inversion"-Prinzips (Mitte) auf die Kopplungsmetriken ist ebenso ersichtlich wie der negative Effekt von Zyklen (rechts).

Zyklische Abhängigkeiten erhöhen nicht nur abstrakt die Werte von Kopplungsmetriken, sondern auch ganz konkret die Auswirkung von Änderungen auf das System. In den seltensten Fällen sind zyklische Abhängigkeiten gewollt, da das Verständnis des Systems dadurch massiv erschwert wird. Diese Komponenten können nur gemeinsam verstanden, getestet und wiederverwendet werden. Es ist nicht möglich, eine Hierarchie aufzubauen und sich entweder Top-down oder Bottom-up in das System einzuarbeiten. Kleine Zyklen sehen noch nicht problematisch aus, in der Praxis sind Zyklen mit mehr als 100 Elementen keine Seltenheit. Visualisiert man diese Abhängigkeiten, entstehen schön anzusehende Bilder wie in *Abbildung 3*, die aber leider nur den existierenden „Big Ball of Mud" *[2]* repräsentieren: Eine klare Struktur ist nicht erkennbar, die Auswirkungen einer Änderung sind kaum abzuschätzen.

Konsistente Struktur auf allen Abstraktionsebenen

Aus dem vorangegangenen Abschnitt ist die Bedeutung der „Kopplung" auf die Änderbarkeit von Software deutlich geworden. Um eine Änderung vorzunehmen, muss der Entwickler aber auch wissen, welcher Code zu ändern ist beziehungsweise wo neuer Code eingebaut werden muss. Das heißt, die Software muss „navigierbar" und sinnvoll zerlegt sein. Die spannende Frage ist, wie das System in Einzelteile zerlegt werden sollte. Hier kommt die im vorigen Abschnitt erwähnte Kopplung ins Spiel: Es wird einfacher, die einzelnen Teile zu verstehen und zu implementieren, wenn man sie isoliert voneinander betrachten kann.

Eine weitere Hilfsgröße ist das Limit von Dingen, die das menschliche Hirn gleichzeitig vorhalten kann. Dieses Limit findet sich auch in anderen Bereichen wieder, beispielsweise in der Organisation von Unternehmen. Sobald ein Teil aus mehr als 15 bis 20 Einheiten besteht, ist eine Gruppierung in einen Container als weitere Abstraktion sinnvoll. Empfehlungen für die Dekomposition eines Systems mittels „Divide and Conquer" wurden bereits vor über 40 Jahren in „Structured Design" *[3]* dargelegt!
Hierbei ist neben der Kopplung auch ein weiteres Prinzip zu beach-

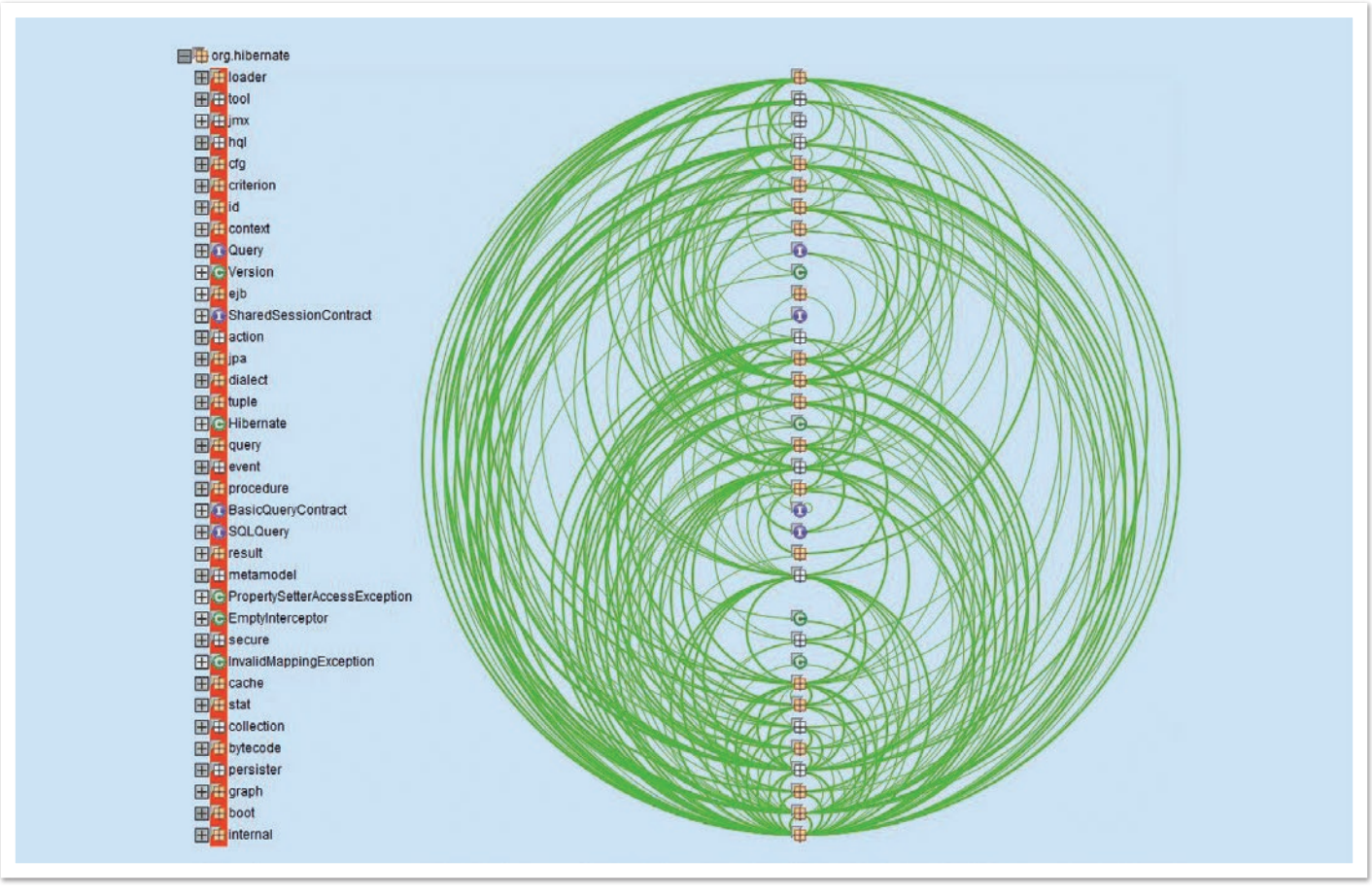

Abbildung 3: Visualisierung von zyklischen Abhängigkeiten (© Ingmar Kellner)

ten, die „Kohäsion“: Als Kohäsion wird dabei die Kopplung von einzelnen Elementen innerhalb eines Containers (zum Beispiel Methoden einer Klasse, Klassen eines Packages) verstanden. Das heißt, in einem Container sollten sich möglichst nur Elemente befinden, die auch miteinander zu tun haben. Bekannt ist dieses Prinzip auch als „Low Coupling, High Cohesion“. Wichtig ist es mir an dieser Stelle zu betonen, dass die Struktur der Software, auch Architektur bezeichnet, einen großen Einfluss darauf hat, wie schnell der Entwickler den Code verstehen und darin navigieren kann.

Das setzt voraus, dass die Elemente der Architektur sich im Code über Namenskonventionen wiedererkennen lassen und das Design der Software auf den verschiedenen Abstraktionsebenen konsistent ist. Es darf auf Code-Ebene keine Abhängigkeit bestehen, die in der Architektur nicht existiert. Ansonsten sind die Abstraktionen ungültig und helfen dem Entwickler nicht beim Verständnis.

Die Strukturierung des Systems ist keine einmalige Aktivität, sonst wäre man wieder beim Wasserfall-artigen „big design upfront“, das ja bereits eingangs als Anti-Pattern erwähnt wurde. Stattdessen muss die Architektur mit dem System „leben“ und laufend angepasst werden, sodass sie bestmöglich den Anforderungen an das System entspricht.

Strukturelle Erosion

Die bisher vorgestellten Konzepte „lose Kopplung“ und „konsistente Struktur“ mögen nicht besonders aufregend klingen und der ein oder andere mag bezweifeln, ob das wirklich relevante Dinge sind, auf die es sich zu achten lohnt. Meine Kollegen und ich haben in den vergangenen Jahren viele Software-Systeme analysiert. Bei den meisten Systemen existiert eine sehr starke Kopplung.

Wie schon erwähnt, werden stark gekoppelte Systeme auch als „Big Ball of Mud“ bezeichnet, da hier wenig von der ursprünglich angedachten Struktur erkennbar ist. Anhand von zwei Systemen möchte ich die Problematik verdeutlichen, dem Modul Hibernate-Core von Hibernate ORM *[4]* und dem Server-Teil der Corona-Warn-App *[5]*.

Analysiert man die Werte für Average Component Dependency (ACD) über einen längeren Zeitraum, lässt sich ein deutlicher Anstieg erken-

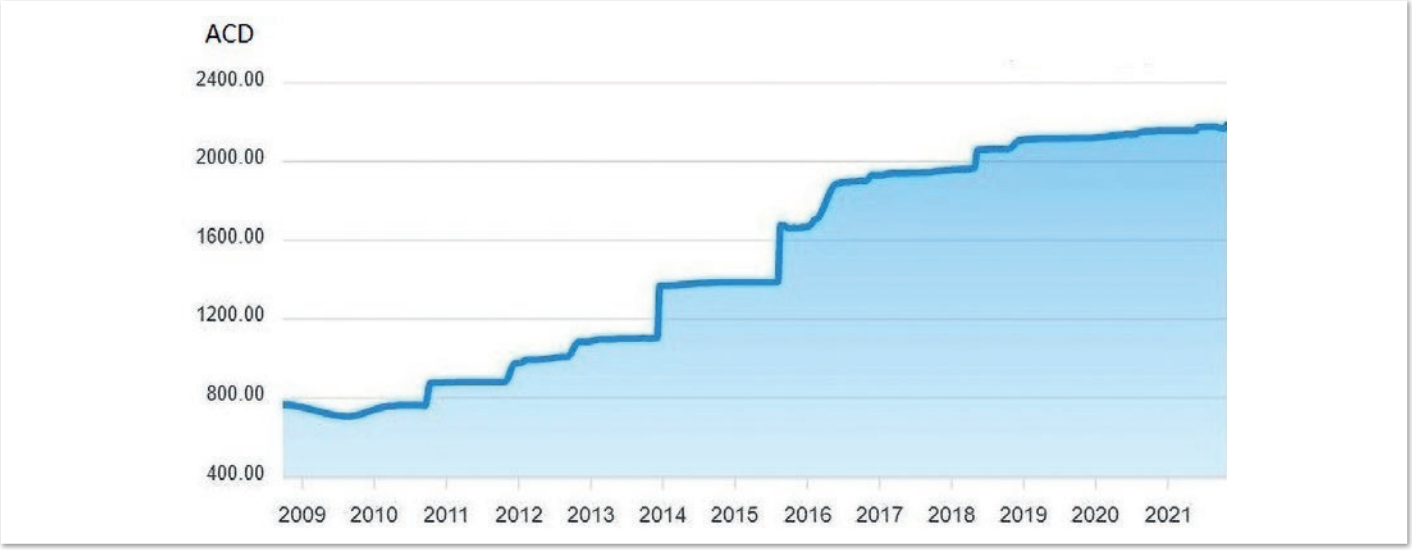

Abbildung 4: Entwicklung des ACD von Hibernate-Core (© Ingmar Kellner)

Abbildung 5: Dateien in der größten Zyklengruppe in Hibernate-Core (© Ingmar Kellner)

nen für Hibernate-Core, wie in *Abbildung 4* dargestellt. In der Version 5.6.1 liegt der Wert bei über 2.100! Das heißt, jedes Source-File im Projekt hängt durchschnittlich von 2.100 anderen Source-Files direkt oder indirekt ab. Bei einer Gesamtzahl von rund 3.500 Source-Files bedeutet dies, dass jede Datei an rund 60 % aller anderen Dateien gekoppelt ist. 90 % des Codes ist in Dateien, die an einem Zyklus beteiligt sind.

Im Release 5.6.1 sind rund 75 % des Codes in einem einzigen Zyklus bestehend aus 2.211 Dateien beteiligt. Auch dieser Zyklus ist kontinuierlich gewachsen, wie in *Abbildung 5* dargestellt, und so dominant, dass die Anzahl der beteiligten Dateien praktisch identisch mit dem ACD-Wert ist. Einen Ausschnitt aus dem Abhängigkeitsgeflecht konnte man bereits in *Abbildung 3* betrachten.

Im Server-Teil der noch jungen Corona-Warn-App lässt sich dieser Trend in kleinerem Ausmaß erkennen. Analysiert habe ich bis zur Version 2.15. Das System ist mit rund 17.000 Zeilen Code überschaubar (Tests und generierter Code ausgeklammert). Auch dort nimmt der an Zyklen beteiligte Code zu und beträgt momentan bereits fast 39 %. Der Code wird kontinuierlich auf Probleme geprüft, doch das verwendete Tool SonarQube bietet seit einigen Jahren keine Zyklenerkennung mehr. In Bezug auf die strukturelle Erosion ist das Projekt folglich blind.

Die Entwicklung dieser beiden Projekte ist ein gutes Beispiel für Software-Entropie, die unter anderem von M.M. Lehman beschrieben wird in „Laws of Program Evolution" *[6]*: „As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

Schauen wir uns an, wie sich die strukturelle Erosion vermeiden lässt.

Best Practice 1: Tooling

Es ist in großen Projekten mit vielen Entwicklern unmöglich, jede Änderung am System manuell zu prüfen. Aus unserer Erfahrung kann die oben genannte Erosion nur durch eine kontinuierliche und automatische Kontrolle erkannt werden. Werkzeuge wie Sonargraph, jQAssistant und andere bieten eine Fülle von Möglichkeiten, um sowohl die Struktur zu überwachen als auch Metriken zu berechnen und Anti-Patterns zu erkennen.

In einer Evaluierungsphase sollte man sich mit den eigenen Bedürfnissen vertraut machen und das Produkt wählen, das diese am besten erfüllt. Werden die Tools für bereits existierende Projekte eingeführt, besteht die Gefahr, dass man von der Menge der gemeldeten Probleme erschlagen wird. Hier bietet es sich an, nach der „Pfadfinder"-Regel („leave the camp ground cleaner than you found it") auf jeden Fall zu verhindern, dass neue Probleme hinzukommen, und die bestehenden dort aufzuarbeiten, wo Code geändert wird. Wichtig ist die kontinuierliche Prüfung und die Unterstützung durch das Management für notwendige Refactorings.

Best Practice 2: Keine Architekturprüfung durch Deployment-Struktur

Die meisten Projekte sind in mehrere Module unterteilt. Wenn für jedes Teil-Element in der Architektur ein eigenes Modul existiert, hat das zwar den Vorteil, dass sich über die erlaubten Modul-Abhängigkeiten keine schwerwiegenden Architekturverletzungen einbauen lassen, die Nachteile sind allerdings nicht zu vernachlässigen:

1. Die große Anzahl der Module erschwert das Verständnis des Systems
2. Die Pflege der Abhängigkeiten in den Modulen ist aufwendig

Noch unübersichtlicher wird es, wenn ein System in eine hohe Anzahl von Microservices aufgeteilt ist. Zwar werden auf diese Weise die Abhängigkeiten im Code auf den jeweiligen Service begrenzt und es können keine Zyklen im Code entstehen, die das gesamte System beinhalten; allerdings bestehen Abhängigkeiten zwischen den Services, die nun jedoch erst zur Laufzeit aufgelöst werden und dadurch noch schwerer nachzuvollziehen sind. Systeme, bei denen das Abhängigkeitsgeflecht nur noch schwer zu durchschauen ist, werden gerne als „Distributed Big-Ball of Mud" bezeichnet. Wir beobachten einen Trend wieder in Richtung größerer Services, die unter dem Label „Self-Contained Systems" *[7]* propagiert werden.

Meine Empfehlung ist, die Architekturprüfung nicht auf die Infrastruktur abzuwälzen! Verwenden Sie passende Werkzeuge, die genau dafür gebaut worden sind, und strukturieren Sie den Code in möglichst wenige Module, so wie es das Deployment später erfordert.

Best Practice 3: Programmier-Prinzipien

Aber nicht nur das passende Tooling ist wichtig. Auch bei der Programmierung selbst lässt sich die Kopplung reduzieren, indem man sich an ein paar Prinzipien orientiert.

Zum einen ist es die Verwendung von unveränderlichen Datenklassen, die seit Java 14 als „Records" direkt unterstützt werden. Zusammen mit „Pure Functions", also Funktionen ohne Seiteneffekte, die lediglich einen Rückgabewert liefern, bieten sie eine saubere Trennung in Datenstrukturen und Funktionen. Das passt meiner Meinung nach gut zu der Aufteilung in Value Objects und Services, die Eric Evans in „Domain-driven Design" *[8]* beschreibt.

Schwer verständliche Abhängigkeiten entstehen auch, wenn Code-Duplikate eliminiert werden, indem eine gemeinsame Oberklasse erstellt wird und der Code durch ein „Extract Method"-Refactoring dorthin verschoben wird. Schon oft habe ich mich selbst für einen durch eine tiefe Hierarchie mäandrierenden Kontrollfluss verflucht! Hier sollte möglichst das Prinzip „favor composition over inheritance" befolgt und der Code in eine separate, unabhängige Klasse ausgelagert werden.

God Classes mit mehreren 1.000 Zeilen Code, die oftmals eine Verletzung des „Single Responsibility"-Prinzips darstellen, lassen sich ebenfalls automatisiert erkennen. Und nicht zuletzt sollte man darauf achten, dass man nur die absolut notwendigen Dinge über Architekturgrenzen hinweg sichtbar macht („Encapsulation/Information Hiding"). Interfaces an Architekturgrenzen zur Implementierung des „Dependency Inversion"-Prinzips wirken wie Brandmauern gegen kaskadierende Änderungen.

Zusammenfassung

Dieser Artikel hat aufgezeigt, dass hohe Kopplung ein Kostentreiber in der Software-Entwicklung ist. Anhand von Beispielen wurde dar-

gestellt, wie sich Kopplung berechnen lässt. Die strukturelle Erosion, das heißt die Ausbreitung von ungewollter Kopplung, ist in vielen Software-Projekten erkennbar.

Eine Kontrolle der Kopplung ist immer notwendig und sinnvoll, egal ob es sich bei der Software um einen Monolithen oder eine verteilte Anwendung bestehend aus Dutzenden Microservices handelt. Ein gut strukturiertes System entsteht nur durch automatisierte Prüfung und regelmäßiges Refactoring, und nicht von selbst durch den Einsatz einer bestimmten Technologie. Grundvoraussetzung ist der Wille, im Team auf diesen Aspekt zu achten. Die Notwendigkeit einer sauberen Struktur wird seit Langem von Experten beschrieben und seit einigen Jahren gibt es dafür die Unterstützung durch passende Tools. Mit der Umsetzung weniger Best Practices lässt sich ein „Big Ball of Mud" zuverlässig vermeiden.

Investitionen an dieser Stelle zahlen sich aus: Einer unserer Kunden berichtet von einer Produktivitätssteigerung der Entwickler von 50 % *[9]*. Und nicht nur das: Es macht einfach mehr Spaß, an einer gut strukturierten Codebasis zu arbeiten, in der Veränderungen schnell realisierbar sind.

Quellen

[1] Lakos, J. (1996): Large-Scale C++ Software Design, Addison Wesley

[2] Foote, B.; Yoder, J. (1999): Big Ball of Mud, *http://www.laputan.org/mud/*

[3] Yourdon, E; Constantine, L (1979): Structured Design, Prentice-Hall

[4] Hibernate-ORM, *https://github.com/hibernate/hibernate-orm*

[5] Corona Warn App (Server), *https://github.com/corona-warn-app/cwa-server*

[6] Lehman, M. M.; Belady, L.A. (1985): Program evolution: processes of software change, Academic Press.

[7] Self-Contained Systems, *https://scs-architecture.org/*

[8] Evans, E. (2004): Domain-Driven Design, Addision Wesley

[9] Baldauf, T.; von Zitzewitz, A. (2015): Case Study Environment Agency Austria, *https://www.hello2morrow.com/whitepapers/50/download*

Ingmar Kellner
hello2morrow GmbH
i.kellner@hello2morrow.com

Ingmar Kellner programmiert seit 20 Jahren, mit einem starken Fokus auf Java. Seit 10 Jahren beschäftigt er sich bei der hello2morrow GmbH mit Code-Qualität und speziell mit der automatisierten strukturellen Analyse. Dort ist er einer der Entwickler des Sonargraph, eines Tools zur statischen Code-Analyse mit Schwerpunkt auf Architekturprüfung, und berät Kunden bei der Umsetzung von Qualitätsverbesserungen.