GraalVM: Neue Serie S. 10

Funqy: Neues Serverless API in Quarkus S. 32

Warum spricht jeder über DDD? S. 74

JAVA Mag

Java magazin™

Java | Architektur | Software-Innovation

JavaFX & GraalVM

In dubio pro Dukeo!

Ausgabe 1.2021

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

4 194586 709801 01

www.javamagazin.de

Architekturpatterns in Modulithen – Teil 2

Wer mit wem reden darf

So manche Codebase macht nur auf den ersten Blick einen aufgeräumten Eindruck. Schön in Packages sortierter Code ohne Abhängigkeitsmanagement ist wie ein „aufgeräumtes“ Kinderzimmer, bei dem einem die Lawine entgegenkommt, wenn man es wagt, die Schranktür aufzumachen. Um zu verhindern, dass Abhängigkeitszyklen und wuchernde Queraufrufe den Code zu einem „Big Ball of Mud“ machen, gilt es, höllisch aufzupassen.

von Arnold Franke

Module in Softwaremonolithen sind toll, das haben wir im ersten Teil dieser Artikelserie gelernt. Das Schneiden von Code in sinnvolle fachliche Kontexte, aus denen man Module bildet, ist dabei schon die halbe Miete zur erfolgreichen Umsetzung dieser Architekturform – aber eben nur die halbe. Um überhaupt in den Genuss ihrer Vorzüge zu kommen, ist es ebenso wichtig, die Interaktionen und Abhängigkeiten der Module sorgfältig zu modellieren.

Schnittstellendesign im Modulithen

Das fängt schon beim Design der Modulschnittstellen an. Welche Operationen soll ein Modul nach außen bereitstellen und in welcher Form gibt es den anderen Modulen seine Daten preis? Ebenso essenziell: Welches Modul darf überhaupt von wem verwendet werden und wie vermeidet man Abhängigkeitszyklen, die die Eigenständigkeit der Module wieder kaputtmachen?

Verliert man diese Fragen aus den Augen, entsteht schnell Wildwuchs im Monolithen. Alles ist auf einmal von jedem abhängig und die Zuständigkeiten zwischen Modulen verschwimmen, womit plötzlich alle Vorteile der modulithischen Architekturform dahin sind. Wartbarkeit und Erweiterbarkeit gehen den Bach runter, unerwartete Nebeneffekte häufen sich. Und das, obwohl man doch so schön geschnittene Module hat. Damit das nicht passiert, gibt es einige Patterns zu befolgen, Antipatterns zu vermeiden und diverse Tools, die einen dabei unterstützen. In diesem Artikel geht es darum, wie man sein Abhängigkeitsmanagement im Modulithen nachhaltig umsetzt. Wir werden uns dabei wieder am Beispiel der Domain „Kino“ entlanghangeln.

Artikelserie

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

Welche Methoden gehören zur Schnittstelle?

Ein wichtiges Konzept in modularer Architektur ist, dass ein Modul die Hoheit über seinen fachlichen Subkontext hat und als einziges bestimmen darf, wie man zugehörige Daten manipuliert. Um das zu ermöglichen, muss das Modul seine interne Logik und Daten verstecken und darf nur Methoden veröffentlichen, die den Clients erlaubte Operationen bereitstellen und die interne Logik in sich kapseln. Wie im ersten Teil der Serie behandelt, ist die einfachste Möglichkeit, das in Java zu bewerkstelligen, das bewusste Setzen der Access Modifier von Methoden – *package private* für Interna und *public* für die Schnittstelle.

Hat man sein Modul intern in Subpackages unterteilt, leidet die Kapselung wieder, da die Subpackages modulinterne Methoden als *public* deklarieren müssen, um sich gegenseitig verwenden zu können. Man hat dann immer noch die Möglichkeit, öffentliche Schnittstellen

durch Konventionen zu definieren. Ein geeignetes Mittel dafür ist das klassische Facade-Pattern [1]. Man erstellt in jedem Modul ein Java-Interface, das alle Schnittstellenmethoden definiert. Die Clientkonvention (die jedes Teammitglied kennen muss) ist dann, dass Zugriffe auf ein Modul nur über dessen Interface erlaubt sind.

Ein häufig anzutreffendes Antipattern nennt sich „Visibility for Testing", das Setzen von privaten Methoden auf *public*, nur um sie testen zu können. Das ist ein Schuss ins Knie, da man sich die schöne Kapselung gleich wieder kaputt macht. Das Bedürfnis, das zu tun, ist oft ein Hinweis auf schlecht gewählte Klassengröße bzw. Methodenabstraktion. Entweder eine private Methode ist Teil einer zu testenden Unit, die bereits eine *public* oder *package private* Methode hat, oder sie gehört zu einer tieferen Abstraktionsebene, die man als eigene, testbare Klasse extrahieren sollte.

Noch gefährlicher wird es, wenn man die Kapselung der Module mit dem Shared-Database-Antipattern umgeht. Es ist in Ordnung, dass alle Module eines Modulithen in dieselbe Datenbank schreiben, solange jedes Modul die Hoheit über seinen eigenen Bereich der Daten hat. Sobald aber zwei Module dieselbe Tabelle beschreiben, besteht wieder die Gefahr von auseinanderlaufender Logik und Konflikten, weshalb der Missbrauch der Datenbank als Schnittstelle dringend zu vermeiden ist.

Wie sehen die Methoden der Schnittstelle aus?

Beim Methodendesign der öffentlichen Schnittstelle eines Moduls ist es selten sinnvoll, einfach nur CRUD-Methoden [2] bereitzustellen. Damit reicht man im Grunde nur die Operationen der Persistenzschicht weiter und gibt den Clients wieder die volle Macht über die Daten. Besser ist es, Business Operation Methods nach den tatsächlichen Operationen zu designen, die auf dem fachlichen Subkontext des Moduls erlaubt sind (Listings 1 und 2).

Man sieht, dass im ersten Fall die Clients volle Macht über die Reservierungsdaten haben und jeder damit seine eigene Version der fachlichen Logik implementiert, was dann zu Inkonsistenzen führt. Im zweiten Fall ist die Logik da implementiert, wo sie hingehört – nämlich im Modul *Reservierung*. So wird die Separation of Concerns [3] eingehalten. Das Modul weiß, wie die Daten zu manipulieren sind und auch, dass noch weitere Dinge zu tun sind – wie hier ein Event verschicken, was die Clients gar nicht wissen können. Die Methode *abholen()* kann man gar nicht falsch benutzen und sie macht den Clientcode leicht verständlich, ohne dass man weitere Erklärung durch Kommentare o. ä. braucht.

Was gibt die Schnittstelle nach außen?

Das letzte Puzzleteil ist die Form, in der ein Modul seine Daten preisgibt. Das i-Tüpfelchen des Schnittstellendesigns ist es, ausschließlich Immutable Objects nach außen zu geben. Die Unveränderbarkeit der Objekte garantiert dem Entwickler an jeder Stelle im Code, dass es sich um originale Daten in validem Zustand vom zuständigen fachlichen Modul handelt, egal welchen Weg durch die Anwendung sie schon hinter sich haben. Insbesondere in der großen Codebase eines Modulithen ist das ein großer Vorteil, der das Fehlerpotenzial nochmal deutlich reduziert.

Üblicherweise gibt es in einem großen Projekt unterschiedliche Sichten auf die Daten. Ein Nutzer im Internet will Daten vermutlich in einer anderen Form präsentiert bekommen als das Backend sie in der Datenbank speichert. Ein fachliches Modul sollte daher eine externe fachliche Abstraktion auf seine Daten bereitstellen, mit denen Clients und andere Module arbeiten können. Metainformationen wie Erstellungszeitstempel oder sogar die Datenbank-ID sind dafür nicht unbedingt relevant, dafür kommen vielleicht weitere Informationen hinzu, die sich aus den Rohdaten ergeben.

Listing 1: Schnittstelle mit CRUD-Methoden

```
// Implementierung der Serviceschnittstelle im Reservierungsmodul
public Reservierung read(Long id) {
  return repository.findById(id);
}
public void update(Reservierung reservierung) {
  repository.save(reservierung);
}

public void delete(Long id) {
  repository.delete(id);
}
// Verwendung der Schnittstelle in Clientmodul 1:
// Reservierung abholen
Reservierung reservierung = reservierungService.read(reservierungId);
reservierung.setAbgeholt(true);
reservierungService.update(reservierung);

// Verwendung der Schnittstelle in Clientmodul 2:
// Reservierung abholen
reservierungService.delete(reservierungId);
```

Listing 2: Schnittstelle mit Business Operation Method

```
// Implementierung der Operation "abholen" im Reservierungsmodul
private void abholen (Long reservierungId) {
  Reservierung reservierung = repository.findById(reservierungId);
  reservierung.setAbgeholt(true);
  Reservierung reservierung = repository.save(reservierung);
  eventPublisher.publishEvent(new ReservierungAbgeholtEvent(reservierung));
}
// Verwendung der Schnittstelle in Clientmodul 1:
reservierungService.abholen(reservierungId);
// Verwendung der Schnittstelle in Clientmodul 2:
reservierungService.abholen(reservierungId);
```

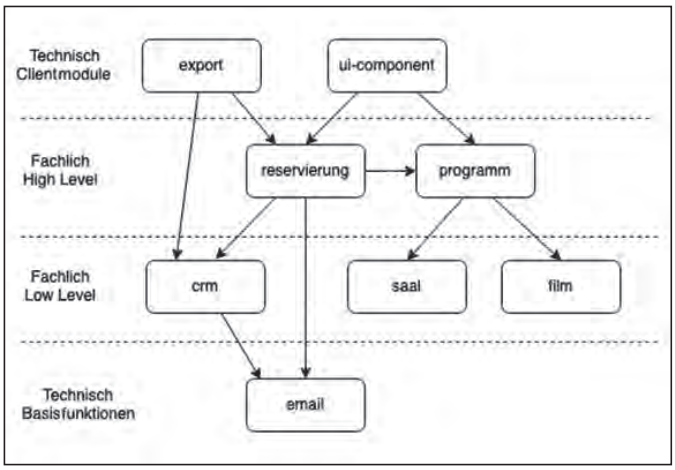

Abb. 1: Unidirektionaler Abhängigkeitsgraph

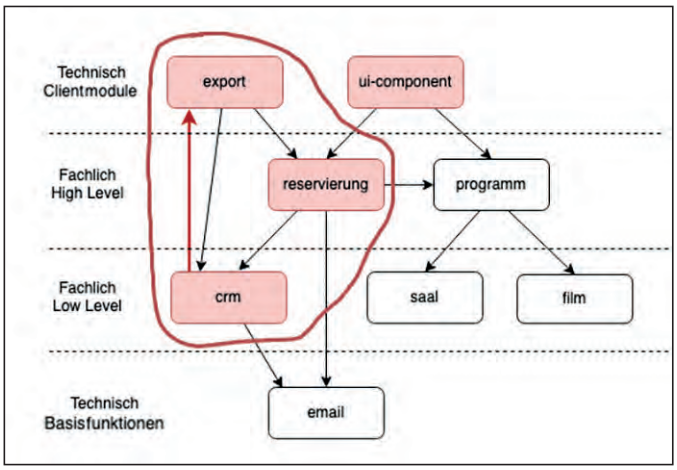

Abb. 2: Eine falsche Abhängigkeit führt zu einem großen Cycle

Dabei muss man höllisch aufpassen, dass man nicht in das Antipattern „Too many layers of abstraction" verfällt. Es gibt Projekte, die für jede technische Schicht und jedes externe System eine eigene Datenabstraktion einführen. Die schiere Menge an Konvertierungscode ist dann zu viel des Guten, und ein Feld in der gesamten Codebase hinzuzufügen, wird zur Mammutaufgabe. Hier schadet die Abstraktion mehr als sie nutzt. Verwende so wenig Abstraktion wie möglich, aber so viel wie nötig: „All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection" [4].

Ein weiteres (Anti-)Pattern im Zusammenhang mit Datenrepräsentation ist „Active Record" [5], wie es in manchen Web Application Frameworks existiert. Dabei hält ein in der Anwendung herumgereichtes Objekt nicht nur Daten, sondern auch zugehörige Logik und vor allem Methoden, die das Objekt direkt in die Datenbank persistieren können. Das bedeutet, dass man z. B. an jeder Stelle im GUI-Code einfach *.save()* auf einem x-beliebig veränderten Objekt aufrufen kann und es wird in der Datenbank aktualisiert. Was in einer kleinen Ein-Tabellen-Anwendung bequem und pragmatisch wirkt, wird in der großen Monolithen-Codebase zum Problem. Es wirft jede Kapselung von Logik über den Haufen, verletzt Single Responsibility und Separation of Concerns und kann zu einem Chaos führen, in dem sehr schwer nachzuvollziehen ist, wann und wo ein Objekt geändert wird.

Abhängigkeitsmanagement zwischen Modulen

Eine weitere Herausforderung, die sich in großen Softwaresystemen stellt, ist die Verdrahtung der Module untereinander. Welches Modul darf welches Modul verwenden? Darüber muss man sich sowohl bei einer Microservices-Landschaft als auch beim Modulithen Gedanken machen. Es ist dabei sinnvoll, sich die Hierarchie der Module bewusst zu machen. Welche fachlichen Module bieten eher grundlegende Funktionalität, welche bauen darauf auf und bedienen eher High Level Use Cases? Welche technischen Querschnittsmodule werden von der Fachlichkeit benutzt und welche müssen die Fachlichkeit kennen? Durch die Beantwortung dieser Fragen ergibt sich ein von oben nach unten gerichteter unidirektionaler Abhängigkeitsgraph, der eine leicht beherrschbare Struktur ins Projekt bringt.

Im Beispiel unserer Kinosoftware könnte man die eher stammdatennahen Module *saal* und *film* sowie das Kundenmodul als Low Level und die darauf aufbauenden *reservierung* und *programm* als High Level identifizieren. *email* steht am Ende des Graphen, weil es von der Fachlichkeit benutzt wird, und die *ui-component* sowie ein Datenexportmodul wären Beispiele für Module, die die Fachlichkeit kennen müssen. Auf einmal entsteht ein aufgeräumtes Bild unserer Anwendungsarchitektur, das man leicht in seinen Kopf bekommt, ohne viel Hintergrundwissen zu haben (**Abb. 1**).

Sehr wichtig ist dabei die Vermeidung von Abhängigkeiten, die in die falsche Richtung fließen. Insbesondere Cyclic Dependencies [6], also Zyklen im Abhängigkeitsgraphen, sind gefährlich. Wenn Module gegenseitig voneinander abhängen, dann macht das den modularen Ansatz kaputt, denn sie sind nicht mehr unabhängig voneinander verwend- und änderbar und bilden im Grunde ein neues Riesenmodul. Nebenwirkungen durch Änderungen in einem einzelnen Modul werden so viel schwerer abschätzbar, denn potenziell können dadurch alle Module im Zyklus und alle Module, die diese benutzen, beeinflusst werden oder kaputtgehen. In unserem Beispiel in **Abbildung 2** ist das deutlich erkennbar, wenn man nur eine falsch gerichtete Abhängigkeit hinzufügt, wie die von CRM auf Export. Dadurch werden die Module *export*, *reservierung* und *crm* zu einem nicht mehr trennbaren Abhängigkeitszyklus. Wenn man jetzt im Modul *export* einen Fehler einbaut, kann dadurch das Modul *crm* kaputtgehen, was plötzlich auch *reservierung* und *ui-component* betrifft. Die halbe Anwendung ist auf einmal miteinander gekoppelt.

Steuern von Abhängigkeiten

Abhängigkeiten zwischen Modulen entstehen üblicherweise durch direkte Aufrufe und die Verwendung von

Klassen eines anderen Moduls. Es gibt aber diverse Möglichkeiten, Abhängigkeiten darüber hinaus zu steuern.

Application Events

Indem man einen direkten Aufruf durch ein Application Event ersetzt, kann man die Abhängigkeit zwischen zwei Modulen umkehren. Ein Modul erzeugt ein Event eines bestimmten Typs, der zum Modul gehört. Ein anderes Modul registriert einen Event Listener, der auf diesen Event-Typ lauscht. Das empfangende Modul hat also eine Abhängigkeit zum sendenden Modul, indem es diesen Event-Typ kennt und nicht umgekehrt. Das ist insbesondere sinnvoll bei Aktionen, die nicht direkt zum fachlichen Use Case gehören, sondern eher Metaaufgaben übernehmen – wie zum Beispiel das Führen eines Auditlogs. In der Java-Welt ist so ein Pattern leicht selbst zu implementieren oder man nutzt z. B. das Application-Event-Feature [7] des Spring Frameworks wie in Listing 3.

Es ist in Modulithen so wie bei Microservices möglich, generell eine Event-getriebene Architektur umzusetzen, in der Interaktionen zwischen Modulen in erster Linie durch Events abgebildet werden. Dabei sind Aspekte wie Asynchronität, Transaktionen, Entkopplung der Module und gesteigerte Komplexität zu beachten und gegeneinander abzuwägen. Ein guter Kompromiss ist die Verwendung von Aufrufen bei direkten Kommandos an ein Modul („Tu bitte das für mich“) und die Verwendung von Events bei Status-Updates („Es ist etwas passiert, wer will, kann reagieren“).

Plug-in Injection

Generell kann Dependency Injection dazu verwendet werden, Abhängigkeiten umzukehren, zum Beispiel im Spring Framework [8]. Modul A definiert ein Interface, das von Modul B als Bean implementiert wird. In Modul A kann man diese Bean jetzt mit dem Typ des Interface injizieren und ihre Methoden aufrufen, ohne das implementierende Modul B zu kennen. So können mehrere Module Plug-ins für die Funktionalität des aufrufenden Moduls schreiben, die dann dort als Liste injiziert und gleichförmig aufgerufen werden.

Listing 3: Application Events

```
// Erzeugung des Events im Reservierungsmodul mit einem Spring Event Publisher:
@Service
public class ReservierungService {
  private ApplicationEventPublisher eventPublisher;
  public void abholen(Long reservierungId) {
// Businesslogik der Methode
    [...]
    eventPublisher.publishEvent(new ReservierungAbgeholtEvent(reservierung));
  }
}
// Reaktion auf das Event im Auditlogmodul mit einem Spring Event Listener
@Component
public class ReservierungAbgeholtEventListener {

  @Eventlistener
  public void onApplicationEvent(ReservierungAbgeholtEvent event)  {
  // Erzeugen des Auditlog-Eintrags aus dem Event
    [...]
  }
}
```

Listing 4: Plug-in Injection

```
// Das Exportmodul definiert über ein Interface, in welcher Form es die
// Daten braucht:
package com.kino.export;
public interface ExportPlugin {
  String export(Long kundeId);
}

// Die fachlichen Module implementieren Plugins und entscheiden damit
// selbst, ob und welche Daten sie exportieren:
package com.kino.crm;
@Component
public class KundeExportPlugin implements ExportPlugin{
  @Autowired
  KundeService kundeService;
  @Override
  public String export(Long kundeId) {
    Kunde kunde = kundeService.findById(kundeId);
    return kunde.getName() + kunde.getTelefonnummer();
  }
}

package com.kino.reservierung;
@Component
public class ReservierungExportPlugin implements ExportPlugin {
  @Autowired
  ReservierungService reservierungService;
  @Override
  public String export(Long kundeId) {
    Reservierung reservierung = reservierungService.findByKundeId(kundeId);
    return reservierung.getNummer() + reservierung.getVorfuehrung();
  }
}

// Die Plugins werden ins Exportmodul injiziert und dort generisch
// aufgerufen. Die Rückgabewerte können verwendet werden, ohne die
// fachlichen Module zu kennen:
package com.kino.export;
@Service
public class ExportService {
  @Autowired
    List<ExportPlugin> exportPlugins;
    public String export(Long kundeId) {
      return exportPlugins.stream()
        .map(plugin -> plugin.export(kundeId))
        .collect(Collectors.joining(","));
    }
  }
}
```

Das ermöglicht zum Beispiel die Umkehr der Abhängigkeiten des Exportmoduls zu den fachlichen Modulen aus **Abbildung 1**, sodass es zu einem Modul der Basisschicht wird, wie Listing 4 zeigt.

AOP

Aspektorientierte Programmierung [9] ermöglicht es, übergreifende Aspekte implizit auszuführen. Dafür definiert man beliebige Join Points, z. B. „alle Methodenaufrufe in Package xy" oder „alle Methoden, die Annotation *@XY* haben", auf die eine übergreifende Logik anzuwenden ist, und Aspects, die die auszuführende übergreifende Logik implementieren. Sogenannte Point Cuts definieren dann, an welchen Join Points welche Aspekte ausgeführt werden. Das Ganze ist so umsetzbar, dass sich die Module, in denen sich die Join Points befinden, und das Modul, in dem sich der Aspect befindet, nicht gegenseitig kennen. Dadurch kann man maximale Entkopplung erreichen, was allerdings auf Kosten der Übersichtlichkeit geht, denn durch AOP implizit ausgeführte Aktionen sind schwieriger nachvollziehbar als direkte Verbindungen.

Tooling

In einem großen Modulithen die Abhängigkeiten aller Module von Hand zu überwachen, ist ein Ding der Unmöglichkeit. Glücklicherweise gibt es eine ganze Reihe Tools, die während aller Phasen des Entwicklungsprozesses das Abhängigkeitsmanagement unterstützen.

IntelliJ Idea und Eclipse

Idea bringt bereits einfache Werkzeuge zur Architekturanalyse mit. Man kann über seinen Code eine Dependency-Matrix generieren [10], die auf Package- und Klassenebene genau aufzeigt, wie viele Abhängigkeiten von wo nach wo bestehen. Verletzungen der Unidirektionalität erkennt man schnell an Abhängigkeiten, die über der Diagonale der Matrix auftauchen. Cycles kann man auflisten und identifizieren, wodurch sie verursacht werden. Für einen ersten Überblick über die Qualität der Codestruktur ist das schon ausreichend (**Abb. 3**; der Cycle aus **Abb. 2** ist sofort erkennbar).

In Eclipse kann man sich verschiedener Plug-ins bedienen, um Abhängigkeiten zu visualisieren und Cycles zu erkennen – wie eDepend [11] oder Java Dependency Viewer [12].

Archunit

Archunit [13] ist eine Testing Library für Architekturregeln. Das API erlaubt es, Codebereiche wie Schichten, Slices etc. zu definieren. Dann kann man Regeln für Zugriffe zwischen den Bereichen festlegen und sie bei jedem Testlauf prüfen. Das eignet sich perfekt zur Definition von Modulen und Festlegung der Abhängigkeitshierarchie zwischen ihnen. Auch Dependency Cycles zwischen Modulen können automatisch erkannt werden, wie Listing 5 zeigt.

Auf Archunit aufbauend hat der Modulithen-Experte Oliver Drotbohm die Spring Boot Erweiterung Moduliths [14] kreiert, die erst vor kurzem in der Version 1.0.0 erschienen ist. Damit kann man sowohl durch einen einfachen Test die grundlegende modulare Struktur eines Spring-Boot-Modulithen sicherstellen (Listing 6) als auch komplexere Abhängigkeiten innerhalb einer Spring-Anwendung testen.

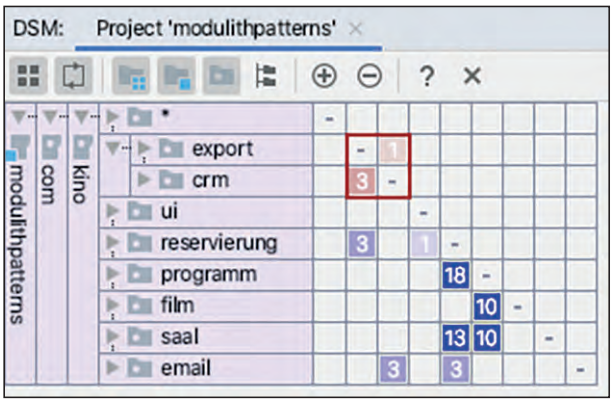

Abb. 3: IntelliJ Idea Dependency Matrix

jQAssistant

jQAssistant [15] ist ein Maven-Plug-in, das eine Neo4j-Graphdatenbank mitbringt. Es kann die Codestruktur

Listing 5: Archunit

```
@Test
void modulesAreFreeOfCycles() {
  JavaClasses classes = new ClassFileImporter().importPackages("com.kino");
  ArchRule rule = slices()
    .matching("..kino.(*)..")
    .should().beFreeOfCycles();
  rule.check(classes);
}
// Mit unserem Dependency Cycle von Abb.2 schlagen die Tests mit diesem Output fehl:
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] - Rule slices matching
                                    ..kino.(*).. should be free of cycles' was violated (1 times):
Cycle detected: Slice crm -> Slice export -> Slice crm
[...]
```

Listing 6: Moduliths

```
@RunWith(SpringRunner.class)
@ModuleTest
public class ModulithsTest {
  @Test
  public void verifyModuleStructure() {
    Modules.of(KinoApplication.class).verify();
  }
}

// Output:
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] - Rule 'slices matching
                                    'com.kino.(*)..' should be free of cycles' was violated (1 times):
Cycle detected: Slice crm -> Slice export -> Slice crm
```

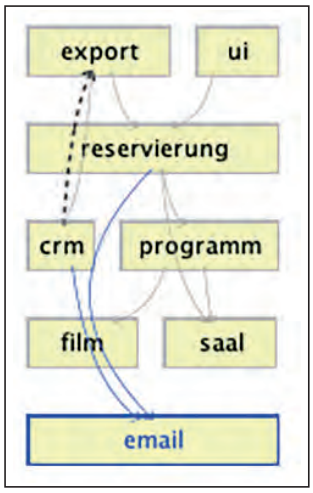

Abb. 4: Structure 101 generiert aus dem Code einen Abhängigkeitsgraph

einer Anwendung in einen Graph einlesen und darauf dann Analysen fahren. Zu prüfende Regeln werden in Cypher definiert, der Query Language für Neo4j. So können neben einfachen Regeln wie Namenskonventionen von Klassen auch komplexe Strukturen auf Modulebene wie Cycles verifiziert werden (Listing 7). Die Einstiegshürde ist etwas höher, weil man sich mit Neo4j und Cypher auseinandersetzen muss, dafür sind die Möglichkeiten zur Definition von Regeln praktisch unbegrenzt.

Structure 101 und Co.

Structure 101 [16] ist eine kostenpflichtige Anwendungssuite für Softwarearchitekten. Das Herzstück ist eine Desktopanwendung, die den Code, den man ihr vorwirft, automatisch analysiert und dann verschiedene Sichten und Auswertungen dafür bereitstellt. Für Modulithenbauer ist besonders interessant, dass man seine Module in der grafischen Oberfläche 1:1 nachmodellieren und daraus einen Soll-Abhängigkeitsgraph bauen kann (**Abb. 4**; unser Cycle ist durch den gestrichelten Pfeil erkennbar). Darin ist die Modulhierarchie definiert, die eingehalten werden muss, Verletzungen werden als Fehler grafisch hervorgehoben. Der Clou ist, dass man diese Soll-Architektur per DIE-Plug-in entwicklungsbegleitend verifizieren kann – Verletzungen werden erkannt, noch während man den Code schreibt. Für die Integration in Continous Integration Pipelines gibt es ein CLI, das den Abhängigkeitsgraph und weitere Regeln verifizieren und bei Verletzungen den Build fehlschlagen lassen kann. Auch beim Refactoring hilft Structure 101, indem man im Architekturdiagramm Klassen und Packages verschieben kann, um zu sehen, wie sich die Änderung auf die Abhängigkeiten auswirkt. In die gleiche Kerbe schlagen auch eine Reihe weiterer Architektursuiten wie Sonargraph/Sotograph [17], Lattix [18], Teamscale [19] und andere.

Fazit

Ebenso wichtig wie der Schnitt der Module sind ihre Interaktionen. Durch geschicktes Schnittstellendesign können wir die Nutzung eines Moduls für Entwickler intuitiv und frei von unerwünschten Nebenwirkungen gestalten. Die Struktur der Anwendung wird durch bewusstes Erzeugen und Vermeiden von Abhängigkeiten zwischen Modulen in einen Zustand versetzt, der es leicht macht, Änderungen an der richtigen Stelle durchzuführen und Auswirkungen zu kontrollieren. Diesen Zustand zu erreichen und zu erhalten, wird durch den Einsatz von bestimmten Patterns und unterstützenden Tools erleichtert, wobei der Nutzen den Mehraufwand schnell überwiegt. Bleibende Herausforderungen sind das Testen der monolithischen Codebase, das Durchführen von weitreichenden Refactorings und die Weiterentwicklung der Anwendungs- und Systemarchitektur eines Modulithen, die im letzten Teil der Serie adressiert werden.

Listing 7: jQAssistent Cypher Query

```
----
MATCH
  (c1:Component)-[:DEFINES_COMPONENT_DEPENDENCY]->(c2:Component),
  cycle=shortestPath((c2)-[:DEFINES_COMPONENT_DEPENDENCY*]->(c1))
RETURN
  c1 as Component, nodes(cycle) as Cycle
----
```

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] Gang of Four – Design Patterns https://archive.org/details/designpatternsel00gamm/page/185

[2] CRUD: https://www.codecademy.com/articles/what-is-crud

[3] Separation of Concerns: https://en.wikipedia.org/wiki/Separation_of_concerns

[4] Wheeler, David: https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

[5] Active Record: Martin Fowler – Patterns of Enterprise Application Architecture

[6] Cyclic Dependencies: https://www.lattix.com/why-cyclic-dependencies-are-bad/

[7] Spring Application Events: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#context-functionality-events

[8] Spring IOC: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans

[9] Aspektorientierte Programmierung: https://de.wikipedia.org/wiki/Aspektorientierte_Programmierung

[10] Idea Dependency Matrix: https://www.jetbrains.com/help/idea/dsm-analysis.html

[11] eDepend: https://marketplace.eclipse.org/category/free-tagging/edepend-graphical-dependency-analysis-tool-340

[12] Java Dependency Viewer: https://marketplace.eclipse.org/content/java-dependency-viewer

[13] Archunit: https://www.archunit.org

[14] Moduliths: https://github.com/odrotbohm/moduliths

[15] jQassistant: https://jqassistant.org

[16] Structure 101: https://structure101.com

[17] Sonargraph/Sotograph: https://www.hello2morrow.com/products/sonargraph

[18] Lattix: https://www.lattix.com

[19] Teamscale: https://www.cqse.eu/en/teamscale/overview/