Java aktuell

iJUG Verbund
www.ijug.eu

Jakarta EE
So geht es jetzt weiter

Groovy
Nützliche, weniger bekannte Features

web3j
Ethereum-Blockchain für Java-Applikationen

Jakarta und der große Knall

Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
Ausgabe 05/2019
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 05

eclipsecon Europe 2019

OSGi™ Community Event 2019

Open Innovation and Collaboration

21. - 24. Oktober 2019
Ludwigsburg

Keynotes

Jan Leuridan
Sr. VP, Simulation and Test Solutions PLM Software
Siemens

Matt Rutkowski
CTO Serverless Technologies
IBM

Auszug Sprecherliste

Jens Reimann
Principal Software Engineer
Red Hat

Kamesh Sampath
Director of Developer Experience
Red Hat

Martin Lippert
Principal Software Engineer
Pivotal Software

Die Open Source Software Konferenz für Eclipse Technologien, offene Innovation und Industriekollaboration

Interesse an Cloud Native Java wie Jakarta EE, MicroProfile und Cloud IDEs, dem Eclipse IoT Stack und den erprobten Eclipse Technolgien wie der Eclipse IDE, RCP und Modeling? Auf der EclipseCon vernetzt Ihr Euch mit den Experten!

Frühbucherrabatt bis 1. Oktober - Ab 775 € (+ MwSt.) Anmeldung unter: www.eclipsecon.org

Functional Core für einen seiteneffektfreien Anwendungskern

Thomas Ruhroth, Msg Systems und Kai Schmidt, selbstständig

In Programmiersprachen vermischen sich mehr und mehr funktionale, imperative und objektorientierte Aspekte. Java erhält Funktionalitäten wie Lambdas, Streams oder ein Flow-API, die funktional ausgerichtet sind. JavaScript dagegen erhält Konstrukte wie Klassen und Interfaces und scheint sich von dem Fundament der Prototype-Struktur abzukapseln. Man hört von Vorteilen funktionaler Programmierung, was die Lesbarkeit, Testbarkeit und die Fehlersuche in der Anwendung betrifft. Dennoch bergen funktionale Programmiersprachen schwer zu verstehende und schwer greifbare Konzepte wie Monaden. Wie kann man, wenn man imperativ aufgewachsen ist, einen Einstieg in diese Welt finden und mit dem Wandel der Programmiersprachen mitgehen? Wie nutzt man die Vorteile der imperativen und funktionalen Welten und wie setzt man sie in der Anwendungsarchitektur ein? Eine Möglichkeit ist der Architekturstil „Functional Core/Imperative Shell", der imperative Objektorientierung von funktionalen Elementen der Anwendung trennt.

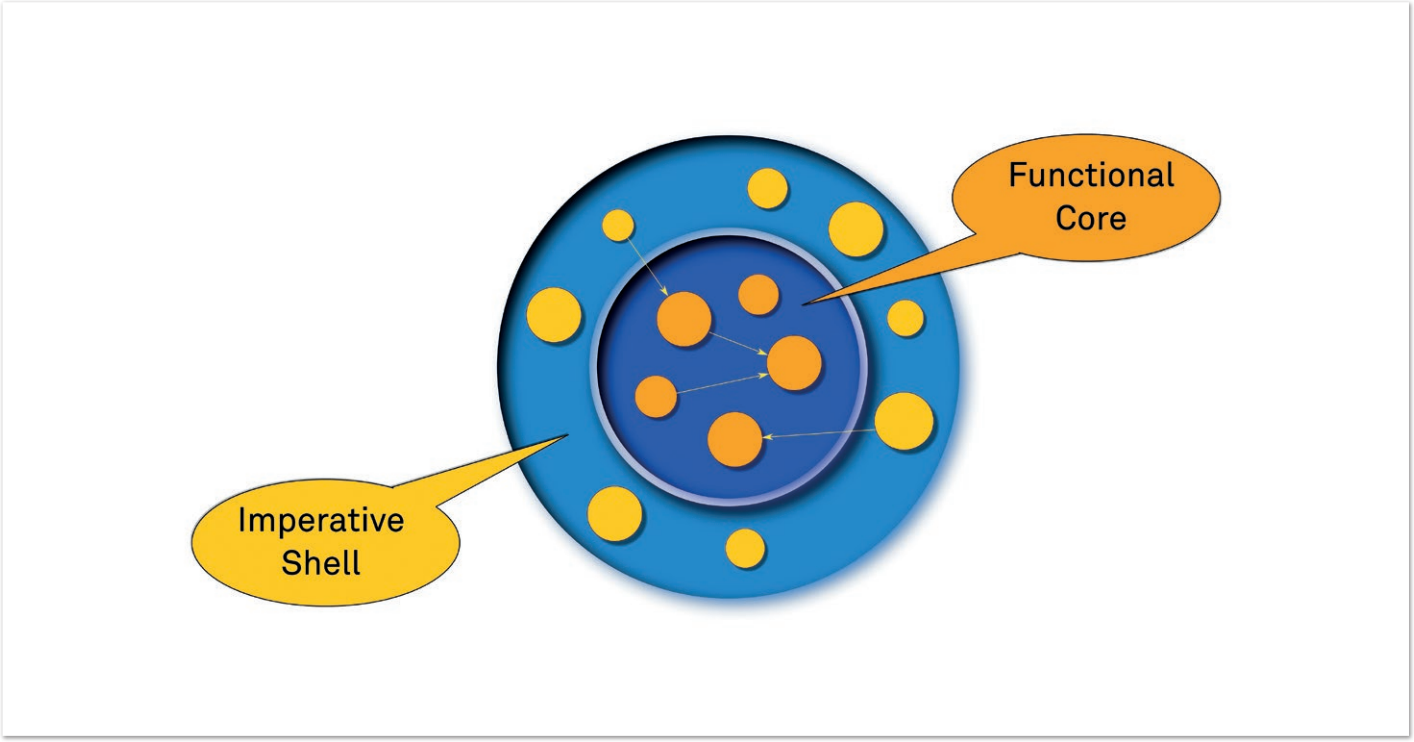

Abbildung 1: Architektur-Schema Functional Core/Imperative Shell (Quelle: Kai Schmidt)

Objektorientierte und funktionale Prinzipien scheinen sich zu widersprechen. Funktionen als Kern der funktionalen Welt sollen keinen Zustand und keine Seiteneffekte haben, wohingegen diese Prinzipien zum Kern der Objektorientierung gehören. Beide lassen sich jedoch sehr elegant vereinen. Objekte lassen sich sehr gut in funktional aufgebauten Methoden verwenden. In diesem Artikel möchten die Autoren zeigen, wie gerade die Kombination dieser beiden Welten entscheidende Vorteile in der eigenen Anwendungsentwicklung haben kann.

Wesentliche Grundlagen funktionaler Programmierung

Während die Objektorientierung als Idee auf der Gruppierung von Daten und direkten Datenmanipulationen aufbaut, ist die wesentliche Idee der funktionalen Sprachen, alle Programmfunktionen als Funktionen aufzufassen (referenzielle Transparenz). Komplexe Abläufe können durch Rekursion und Pattern Matching gestaltet werden. Zentral sind die `puren` Funktionen, die zwei Regeln einhalten müssen:

- Gleiche Eingabewerte liefern das gleiche Ergebnis
- Sie haben keine Seiteneffekte

Die gleichen Ergebnisse können nur sichergestellt werden, wenn die Funktion frei von Zuständen ist. So werden weder Daten aus der Datenbank gelesen noch dürfen die Eingabewerte bei der Verarbeitung verändert werden. Mit Immutable-Klassen stellt Java sicher, dass ihre Instanzen durch eine Funktion nicht verändert werden. Beispielsweise ist jedes String-Objekt „immutable", bei der mittels Funktionen neue Instanzen der Klasse „String" entstehen – die eigentliche Instanz bleibt unverändert.

Seiteneffekte beschreiben Effekte, die Aktionen außerhalb der eigentlichen Berechnungslogik ausführen. Das beinhaltet verschiedene Arten, angefangen bei Veränderungen von Variablen in Objekten bis hin zu Änderungen in I/O-Streams oder in einer Datenbank.

Functional Core/Imperative Shell

Die Idee von Functional Core/Imperative Shell ist es, die Regeln der funktionalen Programmierung für einen Teil der Anwendung zu verwenden (Functional Core), während in dem anderen Teil die zustandshaltenden und seiteneffektbehafteten Techniken aufzufinden sind (Imperative Shell). *Abbildung 1* zeigt, wie die Imperative Shell die Anwendungsfunktionalität unter Ausnutzung des Functional Core bereitstellt. Eine Abhängigkeit des Core auf die Shell ist nicht zulässig. Dieser in der Java-Welt weitgehend noch unbekannt zu scheinende Stil wurde von Gary Bernhardt bereits 2012 auf der RubyConf *[1]* vorgestellt.

Über eine Beispiel-Anwendung zur Vortragsverwaltung einer Konferenz soll dieses Konzept veranschaulicht werden. Der vollständige Quellcode ist unter GitHub *[2]* verfügbar und wird hier auszugsweise und teilweise in abgewandelter Form in den Listings verwendet.

In *Listing 1* ist die Funktion `addNewTalk` dargestellt, die einer Liste (`list`) einen Vortrag (`talk`) nur hinzufügt, wenn dieser nicht `null` ist. Das Erstellen einer neuen, veränderlichen Liste (`mList`) stellt sicher, dass die übergebene Liste (`list`) nicht verändert wird und die Methode somit seiteneffektfrei ist. Die Wertgleichheit des Resultats ergibt sich hier daraus, dass es ausschließlich von den Funktionsparametern abhängt. Die Rückgabe von `List.of` gibt eine Immutable-List zurück, sodass weitere Veränderungen an dieser neuen Liste ausgeschlossen sind.

Die Imperative Shell benutzt den Functional Core, um daraus die Anwendung zu orchestrieren. Sie kann beispielsweise für die Haltung des Anwendungszustandes in Form der vom Core erhaltenen Objekte zuständig sein.

```
public List<Talk> addNewTalk(List<Talk> list, Talk talk){
     var mutableList mList = new ArrayList<Talk>(list)
     if (talk != null) {
        mList.add(talk);
     }
     return List.of(mList);
}
```

Listing 1: Beispiel-Methode im Functional Core

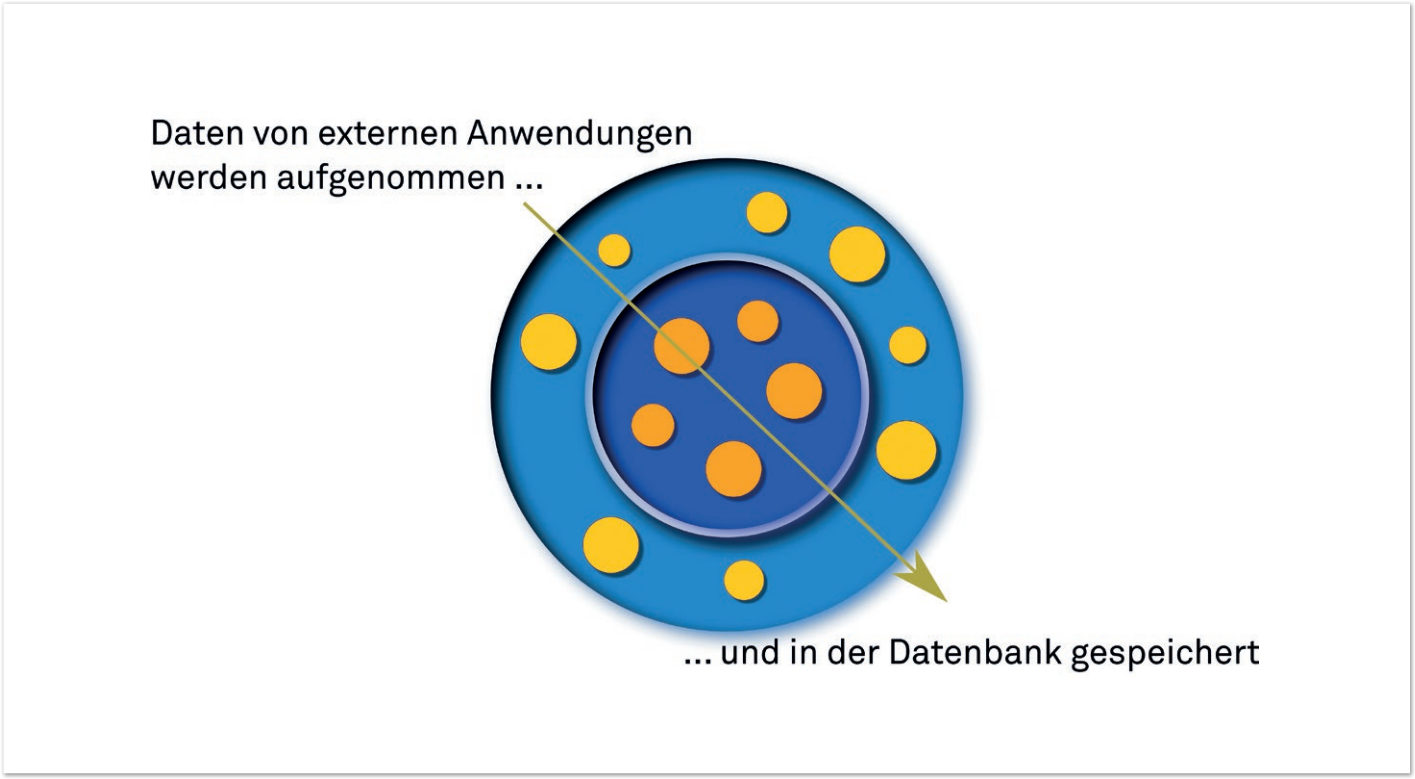

Abbildung 2: Integrationstests als Durchstich (Quelle: Kai Schmidt)

In *Listing 2* wird eine zu *Listing 1* ähnliche Funktion auf einem komplexen Agenda-Objekt ausgeführt. Der neue Zustand nach Ablauf der Methode ist abhängig vom vorigen Zustand. Zusätzlich wird durch das Persistieren auf die Datenbank die Systemumgebung verändert.

Durch die Ausnutzung der Objektorientierung kann eine vereinfachte Schreibweise eingeführt werden. Dabei profitiert man davon, dass das Objekt, das in der rein funktionalen Schreibweise als Parameter übergeben werden muss, bereits die zugehörigen Daten hält. Da es „immutable" ist, kann man direkt auf den State zugreifen und gleichzeitig sicherstellen, diesen nicht zu verändern. So verkürzt sich die Signatur aus *Listing 2* und damit auch dessen Nutzung von `agenda = agenda.addNewTalk(agenda, talk);` zu `agenda = agenda.addNewTalk(talk);`.

Gary Bernhardt nennt dieses Vorgehen in Anlehnung an OO (Objektorientierung) daher FauxO (vom französischen „faux") – also falsche beziehungsweise künstliche Objektorientierung: Die Daten sind zwar nicht vom Code getrennt, es findet jedoch keine Änderung eines Übergabewertes statt.

Testbarkeit im Kern

Die Tatsache, dass der Functional Core nach funktionalen Paradigmen aufgebaut ist, lässt sich für das Testen leicht und in angenehmer Weise nutzen. Bei gleichen Eingabewerten erhält man dasselbe Resultat. Seiteneffekte sind ausgeschlossen und können daher für einen Test nicht relevant sein. Nach dem typischen Muster von „Arrange, Act, Assert" sind die Tests alle gleich aufgebaut: Sie bereiten die Eingabeparameter vor, rufen die zu testende Logik auf und prüfen die Ausgabewerte.

```
public void addNewTalk (Talk talk) {
     agenda = agenda.addNewTalk(agenda, talk);
     someFancyORM.persist(ListEntity.from(agenda.talks));
}
```

Listing 2: Beispiel-Methode der Imperative Shell

```
public class Agenda_When_adding_new_talk {
        @Test
     public void If_talk_does_not_exist_Then_it_is_created() {

     //Arrange
     Agenda agenda = Agenda.initializeAgenda();
     final String topicName = "NewTopic";

     //Act
     agenda = agenda.addNewTalk(topicName);

     //Assert
     AssertThat(agenda.getTalks().iterator().next()
     .getTopic()).isEqualTo(topicName);
     }
}
```

Listing 3: Functional Core Test

```
public class TalkController_When_new_talk_is_created {
     private MockMvc mockMvc;

     @Test
     public void Then_talk_is_shown_in_list() throws Exception
     {

     //Arrange: Neuer Talk ist noch nicht vorhanden
     String newTalk = "TestTalk";
     MvcResult result = mockMvc.perform(get("/"))
                               .andReturn();

     assertThat(result
                     .getResponse()
                     .getContentAsString()
                     .contains(newTalk))
                  .isFalse();

     //Act: Neuen Talk erstellen
     mockMvc
        .perform(post("/talk")
           .content("Vortragsthema=" + newTalk);

     //Assert: Prüfen ob Talk angezeigt wird
     mockMvc.perform(get("/"))
                     .andExpect(status().isOk())
                     .andExpect(content()
                     .string(containsString(newTalk)));
     }
}
```

Listing 4: Imperative Shell Test

Wer bereits Tests für Redux-Reducer oder EventSourcing-Anwendungen geschrieben hat, weiß diesen einfachen Aufbau bereits zu schätzen: Ein bestehender State wird über eine Action beziehungsweise ein Event in einen neuen überführt. Der Test bereitet die Eingabewerte vor, löst das Ereignis aus und gleicht das Resultat mit dem erwarteten Ergebnis ab. Einen beispielhaften Test für den Functional Core zeigt *Listing 3*.

Wie solide ist die Hülle?

Der Test des Functional Core gestaltet sich einfach, jedoch ist die Imperative Shell voll von Seiteneffekten. Die Tests des Functional Core decken glücklicherweise bereits weite Teile (die meisten berechneten Werte) der Anwendung ab. Was fehlt, ist die Sicherstellung der gewollten Seiteneffekte. Folglich müssen noch die Verantwortlichkeiten der Imperative Shell sichergestellt werden; sie hat im Idealfall keine Verzweigungen im Code. Diese sind häufig bereits in den Functional Core gerutscht. Daher müssen nur lineare Pfade

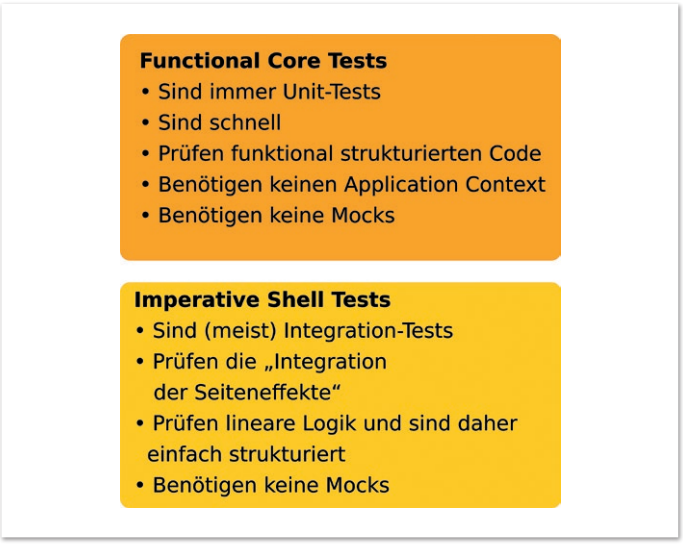

Abbildung 3: Core und Shell Tests im Vergleich (Quelle: Kai Schmidt)

durch die Imperative Shell getestet werden – bei Bedarf mit parametrisierbaren Eingabewerten. Diese Tests kann man wie einen Durchstich durch die Anwendung sehen *(siehe Abbildung 2)*. *Listing 4* zeigt beispielhaft einen solchen Test.

Tests des Functional Core entsprechen charakteristisch Unit-Tests, da es ein Ziel des Functional Core ist, unabhängig von der umgebenden Infrastruktur zu laufen. Demgegenüber stehen die Tests der Imperative Shell, die Integrationstests darstellen. Diese testen beispielsweise, ob Benutzereingaben des Benutzers korrekt in der Datenbank landen und wieder korrekt auf der UI widergespiegelt werden. Sie weisen daher die in *Abbildung 3* genannten Charakteristiken auf.

Die Jagd nach Fehlern

Als Programmierer stellt man fest, dass schwer nachvollziehbare Fehler zu einem hohen Anteil vom Zustand der Anwendung abhängig sind. Häufig lässt sich in imperativen Sprachen nur schwer feststellen, wo es zu Zustandsänderungen innerhalb der Anwendung kommt. Finale Objektreferenzen schützen ausschließlich vor Änderung der Referenz selbst – jedoch nicht vor Veränderung ihrer Inhalte. Diese Änderbarkeit kann praktisch sein, hat aber seine Schattenseiten: Man verliert Hinweise darüber, an welchen Stellen im Code Zustandsänderungen herbeigeführt werden.

Durch die Trennung der Anwendung in funktionale und imperative Bereiche lässt sich diese Eigenschaft einschränken: Allein durch die Lage des Codes innerhalb der Anwendungsarchitektur (also im umgebenden Java-Package) ist erkennbar, wo Zustandsänderungen vorkommen. Innerhalb des Functional Core erleichtert dies die Fehlersuche. Im Programmcode selbst kann man nachvollziehen, wo Änderungen an den Argumenten durchgeführt wurden. Im Debugger kann man deren Werte im Stacktrace zurückverfolgen und die Stellen betrachten, an denen Werte eventuell fehlerhaft zugewiesen wurden.

Lesbarkeit

Durch die Anwendungsarchitektur Functional Core/Imperative Shell ist jede Anwendung in zwei Bereiche unterteilt. Beide Bereiche haben ihre eigenen Charakteristika, sodass man schnell zuordnen kann, in welchem Bereich sich ein Logik-Bestandteil befindet oder wo welche Anpassungen (beispielsweise durch Feature Requests) durchgeführt werden müssen. Außerdem lässt sich im Functional Core durch die Eigenschaften der Unveränderlichkeit der Programmfluss leichter nachverfolgen, wie wir im Abschnitt über die Auffindbarkeit von Fehlern betrachtet haben.

In den Abschnitten über die Testbarkeit konnten wir feststellen, dass die geschriebenen Tests nicht nur die Funktionalität der geschriebenen Anwendung sicherstellen, sondern den Anwendungscode gleichzeitig dokumentieren. Sie teilen sich nahezu automatisch in Unit-Tests und Integrationstests und beschreiben diese meist ohne Zuhilfenahme von Abkapselungstechniken, die zum Beispiel über Mocking-Frameworks erreicht werden. So bleiben Tests leicht nachvollziehbar und konzentrieren sich auf das Wesentliche.

In den funktionalen Teilen der Anwendung dokumentieren bereits die Signaturen der Methoden den Quellcode. Man findet keine Methoden, in denen sich ein „void" in der Methodensignatur eingeschlichen hat. Da man Seiteneffekte ausschließt, würden diese keinen Mehrwert darstellen – welche Aufgabe sollten diese Methoden schon erledigen? Die Tests der Imperative Shell dokumentieren typische Anwendungsflüsse (oder sogar Use Cases), beispielsweise vom Dialog über die Datenbank und gegebenenfalls wieder zurück zu den Daten, die über die Benutzerschnittstelle zur Anzeige gebracht werden.

Rand- und Sonderfälle

Wie häufig in Architekturfragen ist Functional Core/Imperative Shell nicht die Antwort auf alle Probleme. Trotz der beschriebenen Vorteile sollte man sich für jede Anwendung überlegen, ob dieser Stil der passende für das aktuelle Projekt ist. Die Entscheidung hängt von vielen Faktoren ab, angefangen von der zu implementierenden Fachlichkeit bis hin zu den Fähigkeiten des Teams oder sogar zur Organisationsstruktur.

Beispielsweise ist die Trennung von Logik für den Functional Core von der Logik für die Imperative Shell nicht so einfach, wie es anfangs klingt. Angefangen hatten die Autoren mit der Annahme, dass die Fachlichkeit Bestandteil des Functional Core ist, während alles, was nicht zur reinen Fachlichkeit gehört, in die Imperative Shell passt. Es blieb unumstritten, dass die reine Fachlichkeit Bestandteil des Functional Core ist. Der praktikablere Ansatz ist jedoch, dass all jenes, das funktional abbildbar ist, in den Functional Core gehört. Dies kann ebenso querschnittliche Aspekte wie beispielsweise Teile der Autorisierung/Authentifizierung betreffen. Auf diese Weise erhöht sich ebenfalls die Linearität des Imperative-Shell-Codes. Eine hohe Diskussionsbereitschaft im Team und eine gesunde Affinität zu Refactorings schärfen die zugehörigen Sinne.

Der Stil eignet sich gut, sobald auf unveränderlichen Datenstrukturen gearbeitet wird, also beispielsweise in CQRS-Systemen *[3]*. Dort ist durch die Trennung der Schreib- und Lese-Seite der Umgang mit unveränderlichen Strukturen natürlicher. Auf der Schreibseite wird eine Entität beziehungsweise ein Aggregat durch die Anwendung der bisherigen Events rekonstituiert. Änderungen am Aggregat werden nicht direkt am Aggregat durchgeführt, sondern führen zu neuen Events, die wiederum unveränderlich in den Append-Only-Event-Store geschrieben werden. Die verändernde Schnittstelle, die basierend auf den Events die Leseseite aktualisiert, wird schmal, sodass die Datenbank direkt (beispielsweise über Insert und Update-Statements) aktualisiert werden kann.

Bei klassischen CRUD-Anwendungen wird häufig mit Entitäten gearbeitet, die basierend auf den Geschäftsregeln und dem Nutzungsfall direkt verändert werden, bevor sie wieder persistiert werden. Jedoch dürfen Entitäten nicht von Operationen des Functional Core verändert werden. Die im Functional Core ermittelten neuen Ziel-Zustände müssen also in einem separaten Schritt mit der Persistierung zusammengeführt werden. Dies geschieht entweder durch das Überschreiben der Werte der Entität oder über Techniken aus dem ORM-Framework, wie einem `merge` in Hibernate. Sicherlich nicht zu empfehlen ist diese Architektur daher bei Anwendungen, die sich ausschließlich um Datenveränderung kümmern, ohne viel Geschäftslogik zu beinhalten.

Wenn man die Aufgaben des Functional Core betrachtet, kommt die Frage auf, ob im entsprechenden Code Log-Ausgaben erlaubt sein sollten. Klar ist das nach der Theorie, dass dies eine Veränderung des Systemzustands ist. Jedoch wird ein Log nicht die Funktionalität verändern, weil es sich aus Sicht der Anwendung wie ein „write-only"-Speicher verhält. In wissenschaftlichen Dokumenten arbeitet man daher häufig mit einer Abschwächung der puren Funktionen: Bei `effektiv` puren Funktionen beispielsweise sind Seiteneffekte erlaubt, solange diese sich nicht auf das Programm selbst auswirken *[4]*.

Fazit

Die vorgestellte Architektur erlaubt es, Vorteile der funktionalen Welt in Java strukturiert zu nutzen. Selbst wenn man sie nur für Teile verwendet, vereinfacht es die Programmierung. Insbesondere die Regeln aus dem Functional Core können auch in anderen Architekturen vorteilhaft eingesetzt werden, wenn man etwa komplexe Daten durch eigene Datentypklassen nach den Prinzipien von Functional Core gestaltet definiert. Gute, aber sehr einfache Beispiele sind im Java-Framework die Klassen String und LocalDate.

Gleichzeitig ist die Architektur kein Allheilmittel und man muss teilweise einige Erfahrungen sammeln und Refactorings durchführen, bis man sich in diesem Stil zurechtgefunden hat. Die Zuordnung von Code in den Functional Core beziehungsweise in die Imperative Shell sowie die Verwendung unveränderlicher Datentypen benötigt eine gewisse Erfahrung. Mindestens in der Anfangszeit wunderte sich wahrscheinlich jeder, wieso ein String nicht verändert wurde, obwohl man die Methode „substring" oder „replace" darauf aufgerufen hatte. Man gewinnt jedoch leicht testbaren Code mit verständlichen und schnellen Unit-Tests und wenigen, von Natur aus langsameren Integrationstests.

Java wurde bereits mehrfach in Richtung funktionaler Aspekte erweitert und wird sicherlich auch zukünftig weitere Fortschritte machen. Strings haben seit Langem als unveränderliche Datenstruktur in Java ihren Platz gefunden: Unveränderliche Dates (Java 8), Immutable Collections im Stream-API (Java 10) und Erweiterungen Richtung Pattern Matching durch die Erweiterung des Switch-Konstrukts (Java 12) bringen Java den funktionalen Vorteilen näher. Dennoch fehlen weitere Konzepte wie beispielsweise Value Types, die bisher noch keine Berücksichtigung in der Sprache gefunden haben.

Wer weiter in das Thema einsteigen möchte, dem sei insbesondere die Link-Liste von Kasper B. Graversen *[5]* empfohlen. Funktionale Erkenntnisse kann man selbst als Javaianer aus dem sehr gut aufbereiteten Tutorial „Try Haskell!" *[6]* gewinnen.

Quellen

[1] *https://www.youtube.com/watch?v=yTkzNHF6rMs*
[2] *https://github.com/electronickai/functional-core-demo*
[3] *https://martinfowler.com/bliki/CQRS.html*
[4] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif and Mira Mezini (2018) „A Unified Lattice Model and Framework for Purity Analyses" ASE '18, ACM
[5] *https://gist.github.com/kbilsted/abdc017858cad-68c3e7926b03646554e*
[6] *https://tryhaskell.org/*

Thomas Ruhroth
Thomas.Ruhroth@msg.group

Thomas Ruhroth ist Lead IT Consultant bei msg systems AG. In seiner industriellen Arbeitserfahrung arbeitete er als Entwickler, Software-Architekt und Business-Analyst in verschieden Bereichen wie Geographische Informationssysteme und Logistik. In der Forschung liegt sein Fachgebiet in der Softwarespezifikation und in der Entwicklung langlebiger Informationssysteme. Der Wissenstransfer aus der Kombination von Forschungsarbeiten mit industrieller Anwendung ist in vielen seiner Projekte eine treibende Kraft.

Kai Schmidt
mail@kai-schmidt.hamburg

Kai Schmidt ist freiberuflicher Software-Entwickler und -Architekt. Zuvor war er bei den IT-Beratungsunternehmen msg systems AG und Capgemini angestellt und in seiner über zehnjährigen Projekterfahrung größtenteils in Java- und C#-Projekten in den Bereichen Logistik, Flugzeugbau sowie Handel tätig. In seinem letzten Projekt konnte er den hier vorgestellten Architekturstil über ein Jahr lang kennenlernen. Heute berät und beteiligt er sich gern an betrieblichen Anwendungssystemen und ist in der JUG sowie für Kids4IT aktiv.

Early Bird
bis 21. Januar 2020

JavaLand 2020

17. - 19. März 2020 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu

Red Hat
OpenShift

The Kubernetes platform for big ideas

Interactive Learning Portal

https://learn.openshift.com

Launch a pre-configured OpenShift instance with an integrated command-line interface using your web browser!

Our guided training scenarios will help you experiment and learn by solving real-world problems using Kubernetes and other advanced, container-centric tooling.

Red Hat

openshift.com