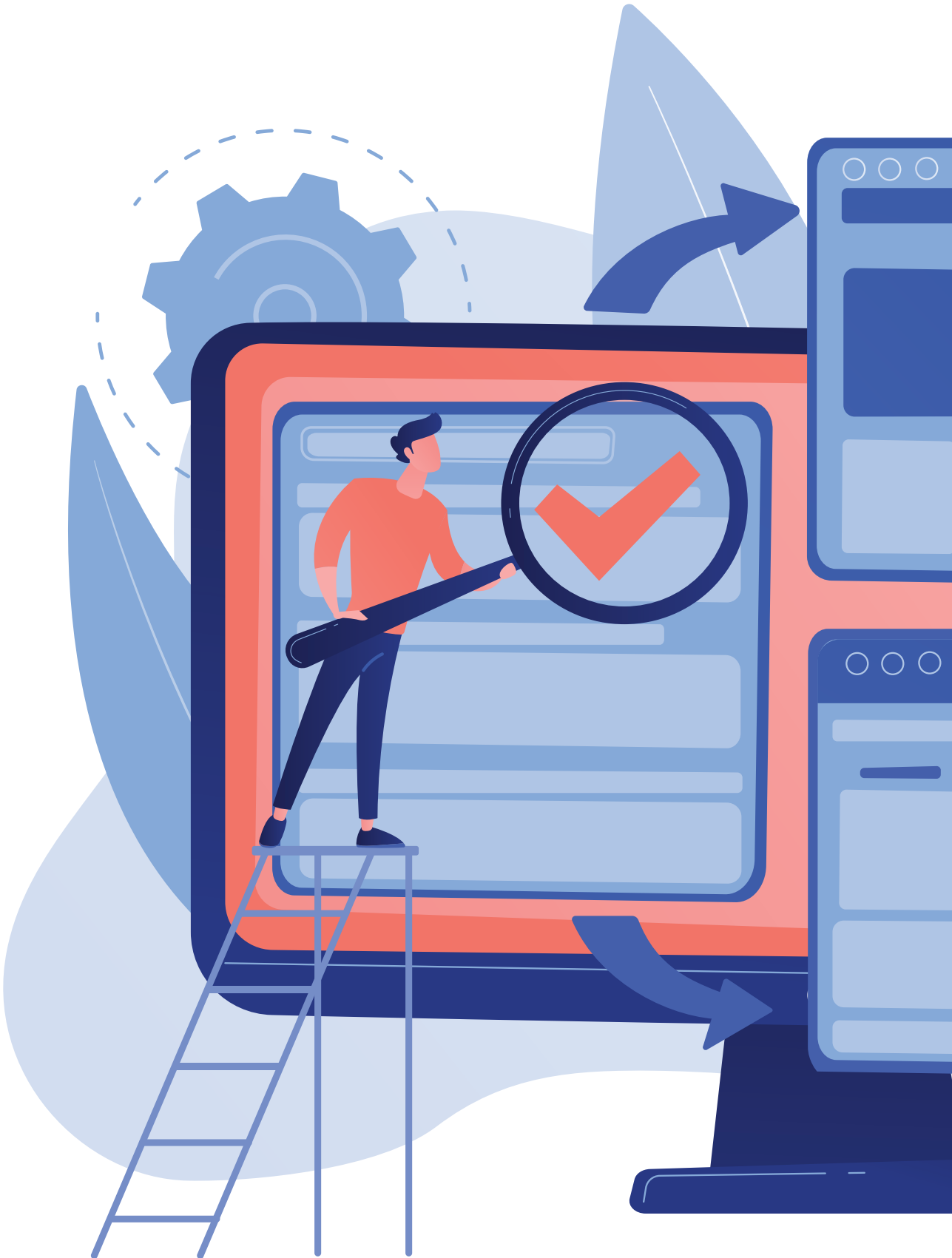
Advanced ArchUnit: Tests auf Bytecode-Analysen aufbauen

Thomas Ruhroth, msg systems, Kai Schmidt



ArchUnit ist ein beliebtes Tool für die Sicherstellung von Architekturvorgaben für JVM-Sprachen. Da die Prüfung im Rahmen von Unittests erfolgt, ist sie für Entwickler einfach und schnell in die lokale Testumgebung und damit in eine Continuous-Integration-Pipeline einzubauen. Grundlegende Regeln, wie Zugriffsberechtigungen zwischen Paketen, lassen sich leicht prüfen. Ebenso liefert ArchUnit konfigurierbare Standard-Architekturen, mit denen man beispielsweise eine hexagonale Architektur sicherstellen kann. Schwieriger wird es, wenn man Prüfungen umsetzen möchte, die (noch) nicht in ArchUnit vorgesehen sind oder die von den bestehenden abweichen.
In diesem Artikel erleben wir einen Ausflug über die verschiedenen Ebenen des ArchUnit-API, um zu verstehen, welche Arten von Prüfungen auf welcher Ebene durchgeführt werden und wie diese Ebenen miteinander kommunizieren.

Einleitung/Motivation

ArchUnit *[1]* ist die bevorzugte Methode der Autoren, in einem Projekt definierte Architektur-Eigenschaften sicherzustellen. Im Gegensatz zu anderen Tools sind die Tests im Code hinterlegt und über Unittests ausführbar. (Eine Einführung ist in *[2]* zu finden.)

Durch viele vordefinierte Prüfungen und deren Fluent-API bietet es einen leichten Einstieg, gute und weitgreifende Architekturtests zu erstellen. So lassen sich eigene Prüfbibliotheken bereitstellen oder Prüfungen durchführen, die einen direkten Zugriff auf das von ArchUnit bereitgestellte Bytecode-Modell benötigen.

Ein Beispiel ist die Prüfung eines im Projekt eingesetzten Architekturmusters namens „Functional Core/Imperative Shell" *[3]*. Das Muster bringt Prinzipien aus der funktionalen Welt in die Architektur objektorientierter/imperativer Sprachen. Da es in ArchUnit derzeit für dieses Muster keine vordefinierten Tests und keine Möglichkeit gibt, dieses durch das derzeitige Lang-API sicherzustellen, war es unser Einstieg in die tiefere Programmierung von Conditions und Libraries für ArchUnit. Das Vorhaben erwies sich als kompliziert, ist daher nicht als Einführungsbeispiel geeignet und würde den Rahmen dieses Artikels sprengen.

Die Dokumentation deckt die Möglichkeiten von ArchUnit nicht vollständig ab – häufig hilft in diesen Fällen ein Blick in den Sourcecode. Dieser Artikel soll den Einstieg in die Erstellung eigener Prüfungen und eigener ArchUnit-Bibliotheken erleichtern. Als Implementierungsbeispiel dient die Verhinderung von als deprecated markierten Aufrufen. Um zuvor ein Grundverständnis für die Architektur von ArchUnit aufzubauen, startet der Artikel mit einem kurzen Überblick.

ArchUnit – Prüfen von Architektureigenschaften

Eine Besonderheit von ArchUnit ist, dass die Analysen auf Bytecode-Ebene durchgeführt werden. Dies bietet den Vorteil, dass alle JVM-Sprachen und Prüfungen nach einem Bytecode-Enhancement unterstützt werden. Auf der anderen Seite bedeutet das, dass Eigenschaften, die sich nicht im Bytecode widerspiegeln, nicht geprüft werden können – beispielsweise generische Definitionen oder Annotationen mit der `RetentionPolicy.SOURCE`.

ArchUnit arbeitet auf drei verschiedenen Abstraktionsebenen *(siehe Abbildung 1)*. Auf der konkretesten Ebene, dem Core-API, hat man den vollen Zugriff auf das Bytecode-Modell und kann Strukturen auf ihre Eigenschaften abfragen. Beispielsweise lassen sich für eine Methode die darin aufgerufenen Methoden ermitteln. Darauf aufbauend zeichnet sich die zweite Abstraktionsebene namens Lang-API aufgrund seines Fluent-Designs mit einer guten Verständlichkeit des mit ihm geschriebenen Codes aus. Ebenso ist dies in der dritten Abstraktionsebene (Library-API) der Fall. Hiermit können vordefinierte Architekturen wie die Onion-Architektur oder Abgleiche mit PlantUML-Modellen definiert werden. Übergreifende Regeln, die beispielsweise zyklische Abhängigkeiten verbieten, befinden sich ebenfalls auf dieser Ebene.

Da das Lang-API unter ArchUnit-Anwendern bekannt sein sollte, führt uns der Überblick zunächst zu dieser mittleren Ebene.

Lang-API

Das Lang-API bietet über eine Art Baukastensystem die Möglichkeit, verständliche Regeln häufig auftretender Prüfungen, wie zum Beispiel von Paketabhängigkeiten, zu erstellen. Eine Regel bildet sich nach einem definierten Schema: `$STRUCTURE that $PREDICATE should $CONDITION`

STRUCTURES können verschiedene Strukturelemente von Java sein, dazu gehören `classes()`, `methods()`, `members()`, `fields()`, `codeUnits()` und `constructors()`. Die STRUCTURE stellt den Einstiegspunkt dar, über den definiert wird, auf welchen Strukturelementen die Prüfung durchgeführt werden soll. Die hierfür zu berücksichtigenden Klassen bauen sich entweder über die Annotation @AnalyzeClasses und die Angabe der Java-Pakete oder im Testcode über einen `ClassFileImporter` auf.

Die Menge von Elementen wird über das PREDICATE auf eine Untermenge gefiltert, auf der die über die Condition definierte Regel geprüft wird. Diese formale Struktur bildet sich direkt in den Umsetzungen ab (siehe *Listing 1* Zeile a). In diesem Fall verwenden wir als STRUCTURE die häufig verwendete Definition `classes()`, um Prüfungen auf Klassenebene auszuführen. Für viele Anwendungsfälle stehen solche Prädikate bereits zur Verfügung. Beispielsweise selektiert die Methode `resideInAPackage` in unserer Verwendung alle Klassen, die in einem übergeordneten Paket „source" enthalten sind – beispielsweise über die Package-Deklaration `com.source.api` oder `de.javaaktuell.source`.

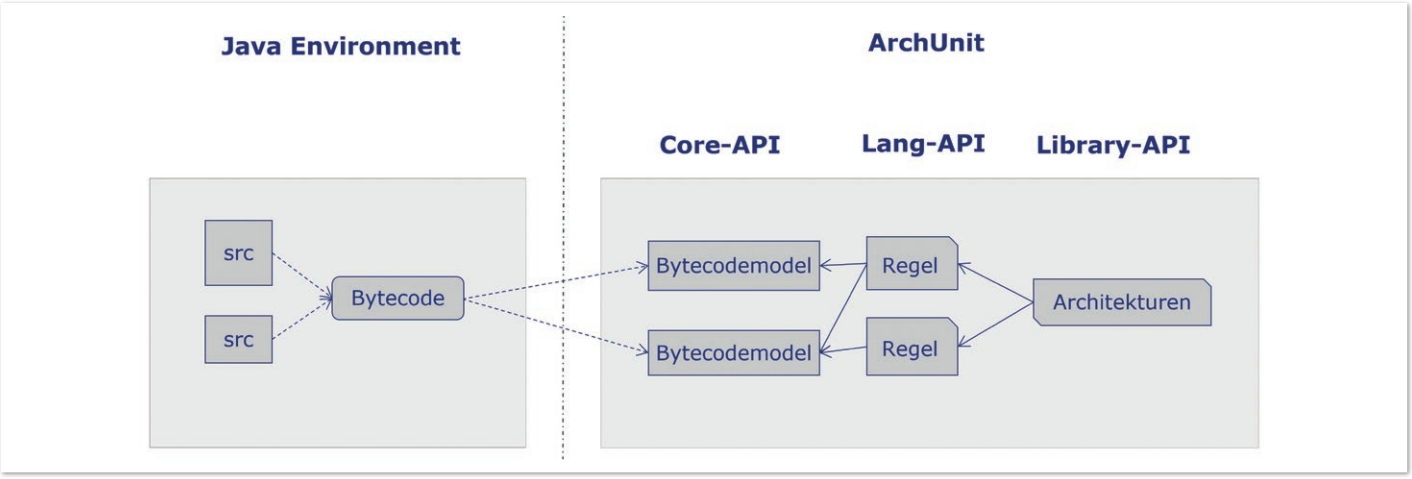


Abbildung 1: Abstraktionsebenen von ArchUnit (Quelle Thomas Ruhroth)

```
a) classes().that().resideInAPackage("..source..")
b) .should().dependOnClassesThat().resideInAPack-
age("..foo..")
```

Listing 1: Beispiel einer ArchUnit Regel im Lang-API

Im zweiten Schritt werden über die CONDITION die Prüfungen definiert, die auf diese Elemente angewendet werden sollen. Auch hierfür stellt ArchUnit vorgefertigte Methoden bereit, die CONDITIONs werden mit der Methode `should` eingeleitet *(siehe Listing 1 Zeile b)*. Über die so zusammengestellte Regel wird definiert, dass jede einzelne Klasse aus `source` von mindestens einer Klasse mit einem übergeordneten Paket `foo` abhängig sein muss.

Library-API

Über die vorgestellte Regel ließen sich die Schichtenzugriffe beliebiger Architekturen definieren. Für gängige Regeln und Architekturen sind derartige Prüfungen bereits auf einer höheren Abstraktionsebene, dem Library-API, gebündelt. Zum Sicherstellen einer Schichtenarchitektur muss konfiguriert werden, wo sich die Schichten befinden und welche Zugriffsmöglichkeiten auf andere Schichten zulässig sind. Da die Zugriffsmöglichkeiten der verschiedenen Schichten für eine hexagonale/Onion-Architektur in ArchUnit bereits vordefiniert sind, genügt in diesem Fall die Zuordnung der Pakete zu den vorgegebenen Schichten *(siehe Listing 2)*.

Während das Lang-API und existierende Architekturprüfungen im Library-API gut dokumentiert sind, findet man zu den Möglichkeiten, die sich nah am Bytecode abspielen, wenig Quellen. Auch im Rahmen dieses Artikels kann keine detaillierte Beschreibung dieses grundlegenden APIs erfolgen. Dennoch soll ein Grundverständnis geschaffen werden, damit das Erforschen des API und weiteres Tüfteln leichter fällt.

```
onionArchitecture()
  .domainModels("architecture.domain.model..")
  .domainServices("architecture.domain.service..")
  .applicationServices("architecture.application..")
  .adapter("persistence", "architecture.adapter.db..")
  // eventuell weitere adapter
```

Listing 2: Beispielverwendung der Onion-Architektur in ArchUnit

Core-API

Den technischen Unterbau für das Lang-API bildet das Core-API, das mit einer Abstraktion des JVM-Bytecodes die Grundlage für alle Prüfungen darstellt. Das Core-API deckt dabei alle Konstrukte ab, die über das Bytecode-Modell repräsentiert sind. Somit ergibt sich eine Repräsentation der Programmstruktur als Graph, auf dem sich die Prüfungen aufbauen lassen.

Einige wichtige Bestandteile sind in *Abbildung 2* dargestellt und folgen dem Sprachschema von Java. So gehören Klassen immer in ein Package. Sowohl die Methode (`JavaCodeUnit`) als auch das Feld (`JavaField`) sind durch ein separates Objekt repräsentiert. Die Benutzungs- und Zugriffsbeziehungen werden durch Objekte vom Typ `JavaMethodCall` beziehungsweise `JavaFieldAccess` abgebildet. Prüfungen auf diesem Objekt-Graph können leicht beschrieben und über eine Definition in Form einer CONDITION an das Lang-API weitergereicht werden. Um diese Struktur besser zu verstehen, sehen wir uns ein einfaches Beispiel an *(siehe Abbildung 3)*: Eine Klasse Agenda befindet sich im Paket `de.test.core` und hat eine Instanzvariable vom Typ `Set` namens `talks`. Auf dieser Variable arbeitet die Methode addTalk, die einen neuen Talk über die `add`-Methode der Klasse `Set` hinzufügt.

Eine – außer als Beispiel – sinnlose Prüfung wäre, sicherzustellen, dass Methoden, die mit `add` beginnen, immer auf ein `Set` zugreifen

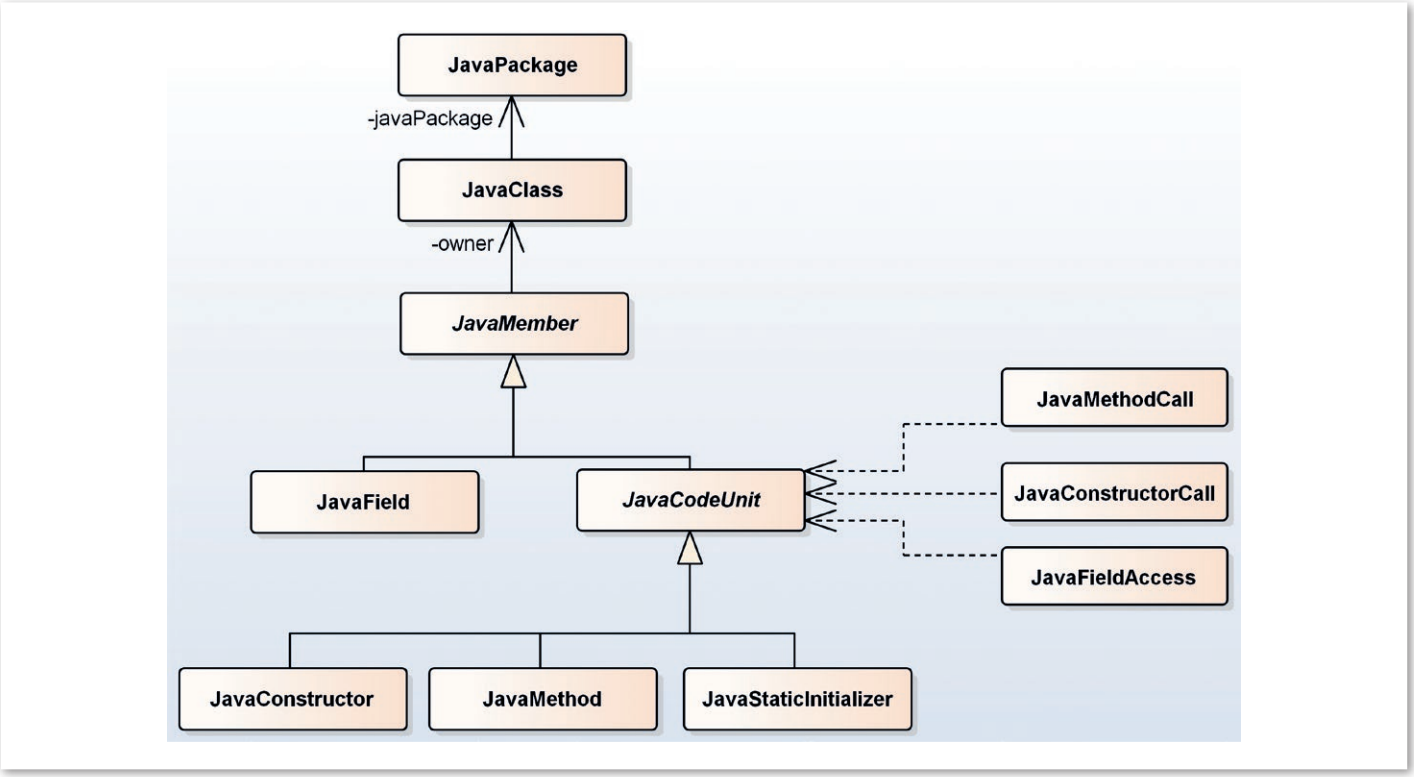


Abbildung 2: Abbildung des Bytecodes im Core-API (Quelle Thomas Ruhroth)

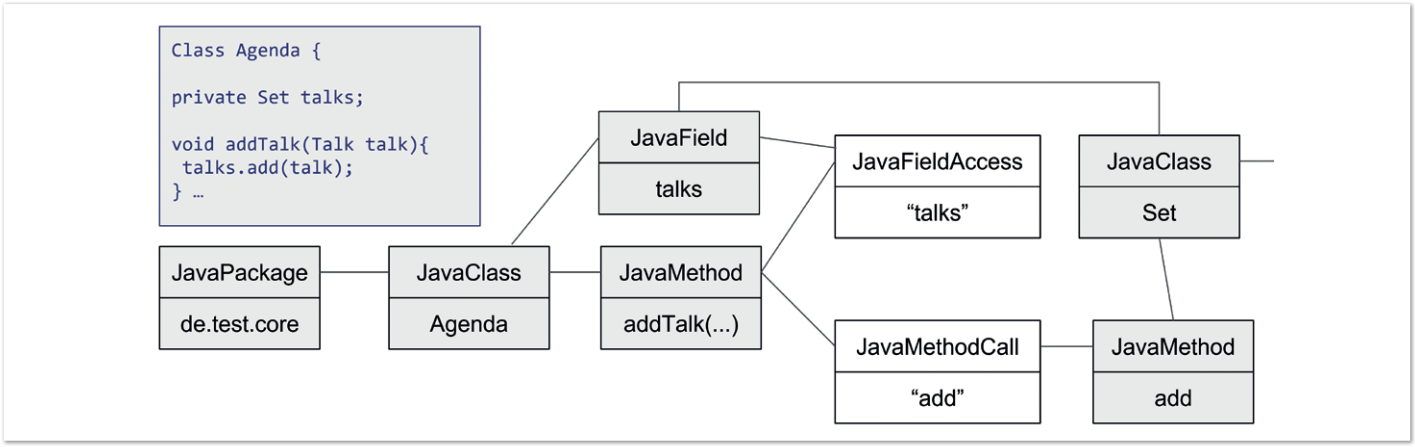


Abbildung 3: Struktur des Core-Modells für ein Codebeispiel (Quelle Thomas Ruhroth)

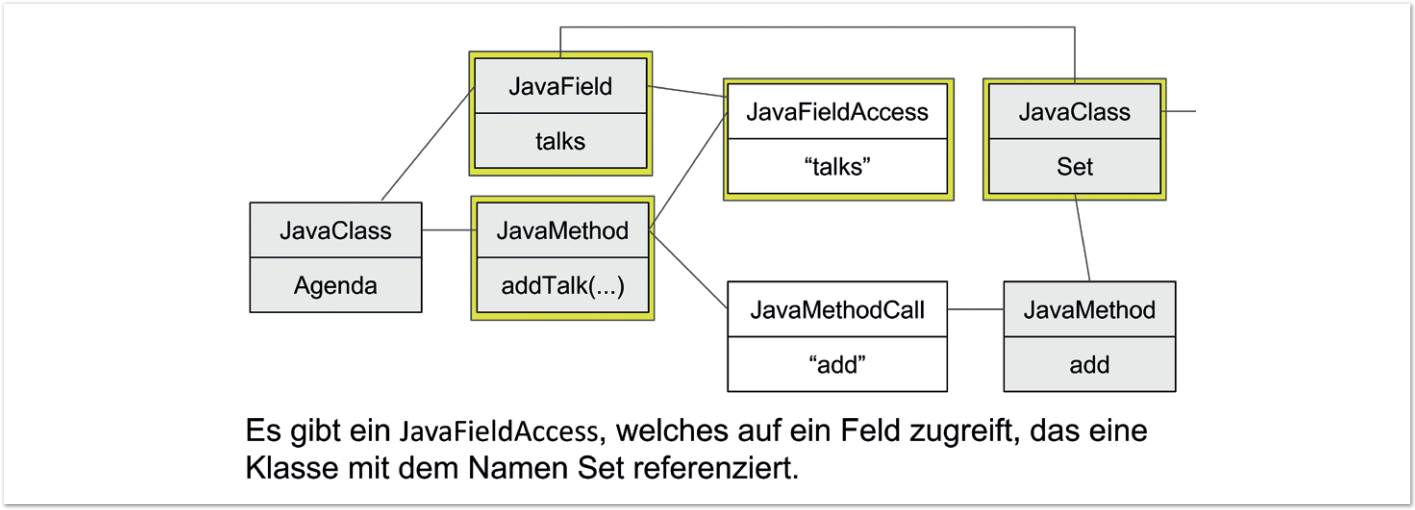


Abbildung 4: Beispiel einer Core-Regelanalyse (Quelle Thomas Ruhroth)

sollen. Über den Graphen lässt sich prüfen, dass für alle Methoden-Objekte, deren Name-Property mit add beginnt, folgende Bedingungen gelten sollen: Es gibt ein `JavaFieldAccess`, das auf ein Feld zugreift, das eine Klasse mit dem Namen `Set` referenziert. Im Graphen *(siehe Abbildung 4)* bedeutet das, dass der Zusammenhang der gelb umrandeten Objekte geprüft wird.

Mit dem Core-API eine Lang-API-Condition erstellen

Mit dem Core-API können Entwickler selbst Prüfungen implementieren und diese anschließend im Lang-API nutzen. Hier werden wir eine vereinfachte Version eines `Deprecated`-Checks aus einem Projekt eines Autors implementieren. Es soll sichergestellt werden, dass Methoden keine als `deprecated` markierten Methoden aufrufen. Ausnahmen bilden Methoden, die selbst als `deprecated` markiert sind. Um dies auf dem Lang-API zu verwirklichen, benötigen wir eine Regel nach dem zuvor vorgestellten Schema. Die Regel müsste eine Form wie aus *Listing 3* aufweisen.

Beim genauen Blick auf das Listing fällt auf, dass im Gegensatz zu *Listing 1* weder nach dem Aufruf von `that` ein Predicate als weitere Methode aufgerufen wird noch beim Aufruf von `should` eine weitere Condition. Das Objekt, das von `that` zurückgegeben wird, kennt keine Methode mit der von uns benötigten Funktion. Ähnlich verhält es sich mit dem Rückgabewert von `should`. Die Rückgabewerte sind Klassen aus dem ArchUnit-Framework. Neue Funktionen für das Lang-API in diesen Klassen könnten über eine Contribution in das ArchUnit-Projekt einfließen. Als Alternative lassen sich selbst implementierte Predicates beziehungsweise Conditions als Parameter von `that` beziehungsweise `should` übergeben.

Der Einstiegspunkt für unsere CONDITION ist das Interface `ArchCondition` *(siehe Listing 4)*. Die zentrale Funktion der `ArchCondition` ist die Methode `check`, die als Parameter die zu prüfende Codestruktur (hier `JavaCodeUnit`) übergeben bekommt. Auf den darauf ausgehenden JavaCalls führen wir die Analyse in der ausgelagerten Methode `checkCall` aus. Das ebenfalls übergebene `ConditionEvents`-Objekt sammelt die Analyseergebnisse.

Für einen `JavaCall` lassen sich sowohl die aufrufende Methode (`getOrigin`) als auch die aufgerufene Methode (`getTarget`) ermitteln. Der Check prüft, ob die aufgerufene Methode oder deren

```
public static final ArchRule DO_NOT_USE_DEPRECATED_OPS
  = methods()
      .that(new NotDeprecatedPredicate())
      .should(new DoNotUseDeprecatedArchCondition());
```

Listing 3: Verwendung von Predicate und Condition in einer ArchRule

```
public class DoNotUseDeprecatedArcCondition extends ArchCondition<JavaCodeUnit> {

 public DoNotUseDeprecatedArcCondition (String description, Object... args) {
   super(description, args);
 }

 public DoNotUseDeprecatedArcCondition() {
   this("not call deprecated CodeUnits");
 }

 @Override
 public void check (JavaCodeUnit codeUnit, ConditionEvents conditionEvents) {
   codeUnit.getCallsFromSelf()
       .forEach(call -> checkCall(call, conditionEvents));
 }

 private void checkCall (JavaCall<?> call, ConditionEvents conditionEvents) {
   AccessTarget target = call.getTarget();

   if (target.isAnnotatedWith(Deprecated.class)
       || target.getOwner().isAnnotatedWith(Deprecated.class)) {
          conditionEvents
          .add(SimpleConditionEvent.violated(
             call.getOrigin(),
             "CodeUnit " + call.getOrigin() +
             " calls deprecated " + target.getFullName()));
     }
  }
}
```

Listing 4: Erstellung einer eigenen Condition

Klasse `deprecated` ist. Auffälligkeiten werden als Architektur-Verstoß in den `ConditionEvents` abgelegt.

Um diese Prüfung, den Architekturtest, über das Lang-API zur Verfügung zu stellen, wird die Prüfung über eine Subklasse `ArchCondition` implementiert. Über den Typ-Parameter geben wir an, dass unsere `check`-Methode auf der Ebene einer `JavaCodeUnit` eingesetzt wird. Die neue Subklasse muss neben der `check`-Methode einen Konstruktor implementieren. Hierbei genügt es, den Super-Konstruktor aufzurufen. Für den leichteren Aufruf lässt sich zusätzlich ein Konstruktor ohne Parameter deklarieren. Nun kann die über das Core-API entwickelte Prüfung in einer normalen Regel innerhalb des Lang-API verwendet werden *(siehe Listing 3)*.

Mit dem Core-API ein Lang-API-Predicate erstellen

Das prinzipielle Vorgehen beim Erstellen eines PREDICATE gleicht dem Vorgehen einer CONDITION. Hier wird jedoch eine Subklasse von `DescribedPredicate` gebaut, die einen Wahrheitswert für die Erfüllung der PREDICATES in der Methode `test` berechnet. Über das Predicate möchten wir ausschließlich Methoden selektieren, die nicht selbst oder über die enthaltende Klasse (Owner) als `deprecated` markiert sind *(siehe Listing 5)*.

```
public class NotDeprecatedPredicate extends DescribedPredicate<JavaCodeUnit> {
  public NotDeprecatedPredicate(String description, Object... params) {
     super(description, params);
  }

  public NotDeprecatedPredicate() {
     this("are not deprecated");
  }

  // Hinweis: Vor ArchUnit 1.0.0 heißt die Methode apply
  @Override
  public boolean test(JavaCodeUnit codeUnit) {
     return !codeUnit.isAnnotatedWith(Deprecated.class)
        && !codeUnit.getOwner().isAnnotatedWith(Deprecated.class);
  }
}
```

Listing 5: Erstellung eines eigenen Predicate

```
@ArchTest
ArchRule doNotUseDeprecatedOps = MyArchLib.DO_NOT_USE_DEPRECATED_OPS
```

Listing 6: Verwendung einer selbst definierten ArchRule

```
@ArchTest
public static final ArchRule configureDeprecatedOpCalls =
  MyArchTestLibrary.deprecationAwareArchitecture()
      .packages("deprecated.callees").areAllowedToBeCalled()
      .packages("deprecated.callers").areAllowedToUseDeprecated();
```

Listing 7: Verwendung mit parametrisierbaren Regeln

```
@Override public EvaluationResult evaluate(JavaClasses classes) {
  EvaluationResult result = new EvaluationResult(this, Priority.MEDIUM);
  DeprecatedPredicate predicate = new
      DeprecatedPredicate(packagesAllowedToUseDeprecated);
  DoNotUseDeprecatedArcCondition condition = new
      DoNotUseDeprecatedArcCondition(packagesAllowedToBeCalled);
          result.add(codeUnits().that(predicate).should(condition).evaluate(classes));
  return result;
}
```

Listing 8: Übergabe der Parameter in das Lang-API

Predicate und Condition im Library-API verwenden

Über verschiedenen (Teil-)Projekte oder Module immer die gleichen Regeln zu erstellen ist mühselig. Meist können die Regeln 1:1 übernommen oder eine zusätzliche in leicht abgewandelter Form erstellt werden. Hier hilft es, Prüfungen über das Library-API zusammenzufassen. Die einfachste Variante ist es, wie die `GeneralCodingRules` aus ArchUnit, Regeln als eine Konstante des Typs `ArchRule` bereitzustellen *(siehe Listing 6)*. Die `ArchRule` enthält die Prüfung, wie man sie auch mithilfe des Lang-API schreiben würde. So lässt sich eine Prüfung der `ArchRule` ohne weitere Definition direkt in `ArchTests` nutzen.

Komplexer wird es, wenn man Regeln parametrisieren möchte. Im folgenden Beispiel gehen wir von einer modifizierten Deprecated-Regel aus, die zwei Arten von Ausnahmen bei der Prüfung erlaubt: In der ersten werden Klassen angegeben, auf die trotz Deprecated-Flag zugegriffen werden darf, ohne dass es zu einer Meldung führt – also eine Art Whitelist der aufgerufenen Methoden. Die zweite Ausnahme ist, dass Pakete als Legacy-Pakete deklariert werden, die auf `deprecated`-Methoden zugreifen dürfen – also eine Art Whitelist der aufrufenden Methoden. Die Parametrisierung soll dabei, wie im Library-API von ArchUnit üblich, als Fluent-API gestaltet sein und die Selektierung über die Paketnamen erfolgen *(siehe Listing 7)*.

Ziel dieser Konfiguration ist der Aufbau zweier Listen: Eine Liste der aufrufbaren Pakete und eine Liste der aufrufenden Pakete, um diese später der `NotDeprecatedPredicate` oder der `DoNotUseDeprecatedArchCondition` übergeben zu können. Ähnlich zu den in ArchUnit bereits auf dem Library-API vorhandenen Architekturen definieren wir eine `DeprecationAwareArchitecture`, um dort die beiden Listen zu deklarieren. Der gleichnamige Methodenaufruf gibt uns genau diese Klasse zurück, sodass wir mit den folgenden Methoden darauf arbeiten können. Vorstellbar wäre es, die Methode `areAllowedToBeCalled` direkt innerhalb der Architektur zu deklarieren. Sie könnte die Paketdeklaration über einen Parameter aufnehmen und in die zugehörige Liste speichern *(ähnlich zu Listing 2)*. Allerdings wäre dies nicht besonders sprechend. Daher wird eine Zwischenklasse verwendet, die über den Aufruf von `packages` zurückgegeben wird und die Deklaration für die Übergabe an die Liste aufnimmt. Diese Zwischenklasse ist als innere Klasse der Architektur definiert und kann daher über die darauf definierten Methoden `areAllowedToBeCalled` und `areAllowedToUseDeprecated` direkt in die Listen der Architektur aufnehmen. Die Rückgabe dieser beiden Methoden ist ebenfalls wieder die Architektur, damit die Konfiguration fortgeführt werden kann. Zusätzlich ist die Architektur eine Subklasse von `ArchRule` und kann somit über die Annotation `@ArchTest` als Unittest ausgeführt werden.

Für die Ausführung der ArchRule von ArchUnit muss die Klasse die über das Interface definierten Methoden implementieren. Darunter auch die Methode `evaluate`. Innerhalb dieser Methode lassen sich nun die Konfigurationsparameter an das Predicate und an die Condition innerhalb ihrer Konstruktoren übergeben, sodass diese mit den hierüber neu gewonnenen Informationen die Selektion und die Prüfung ausführen *(siehe Listing 8)*.

In unserem Beispielcode *[4]* lassen sich der dargestellte Ablauf sowie die weitere Auswertung der übergebenen Paketdeklarationen genauer nachvollziehen.

Fazit

Mit den vorgestellten Techniken lassen sich auf einfache Weise Architekturen für Projekte sicherstellen. In einem Projekt einer der Autoren hat man sich beispielsweise auf eine Auslegung der hexagonalen Architektur geeinigt, die nicht vollständig mit der Definition der von ArchUnit bereitgestellten Onion-Architektur übereinstimmt. Es gibt ein zusätzliches, neben der Architektur stehendes Package zur App-Initialisierung, das in den Adaptern eine zentrale Initialisierung anstößt. Statt diese und einige andere Änderungen in jedem Test neu zu definieren, wurde eine eigene Architektur-Library geschrieben, die neben der angepassten hexagonalen Architektur und dem hier vereinfacht dargestellten Deprecated-Check weitere Prüfungen bereitstellt. Es wird hierdurch eine Vereinheitlichung bei einfacher Einbindung in die Einzelprojekte erreicht.

Quellen

[1] ArchUnit Homepage und Userguide: *https://www.archunit.org*

[2] Peter Gafert 2018: Java-Architekturen dauerhaft sichern mit

ArchUnit. Java Aktuell 02/2018 *https://www.archunit.org/assets/downloads/02-2018-Java%20aktuell-Java-Architekturen_dauerhaft_sichern_mit_ArchUnit.pdf*

[3] Thomas Ruhroth and Kai Schmidt 2019: Functional core für einen seiteneffektfreien Anwendungskern. Java aktuell 05/2019. *https://www.kai-schmidt.hamburg/wp-content/uploads/05_2019-Java_aktuell-Autor-Thomas_Ruhroth_Kai_Schmidt-Functional_Core_fuer_einen_seiteneffektfreien_Anwendungskern.pdf*

[4] Beispielcode zum Artikel: *https://github.com/electronickai/ArchUnit-Layers-Example*

Thomas Ruhroth
msg systems ag
Thomas.Ruhroth@msg.group

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 15-jährigen Projekterfahrung größtenteils in Java und C#-Projekten in den Bereichen Logistik, Flugzeugbau, Finanzen sowie Handel tätig. Er liebt konsistenten Code in den Projekten, auf deren Regeln sich die Teams selbst einigen. Heute berät und beteiligt er sich gerne an betrieblichen Anwendungssystemen und ist in der JUG sowie für Kids4IT und teilweise als Speaker auf Konferenzen und Meetups aktiv.

Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen
43 JUG Saarland

iJUG Verbund

www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Andrisek, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Bildnachweis:
Titel: Bild © freepik
https://freepik.com
S. 10 + 11: Bild © fullvector
https://freepik.com
S. 12 + 13: Bild © freepik
https://freepik.com
S. 20: Bild © storyset
https://freepik.com
S. 24 + 25: Bild © freepik
https://freepik.com
S. 32 + 33: Bild © vectorjuice
https://freepik.com
S. 40 + 41: Bild © rawpixel.com
https://freepik.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

Mediadaten und Preise:
www.doag.org/go/mediadaten

Druck:
WIRmachenDRUCK GmbH
www.wir-machen-druck.de

Inserentenverzeichnis