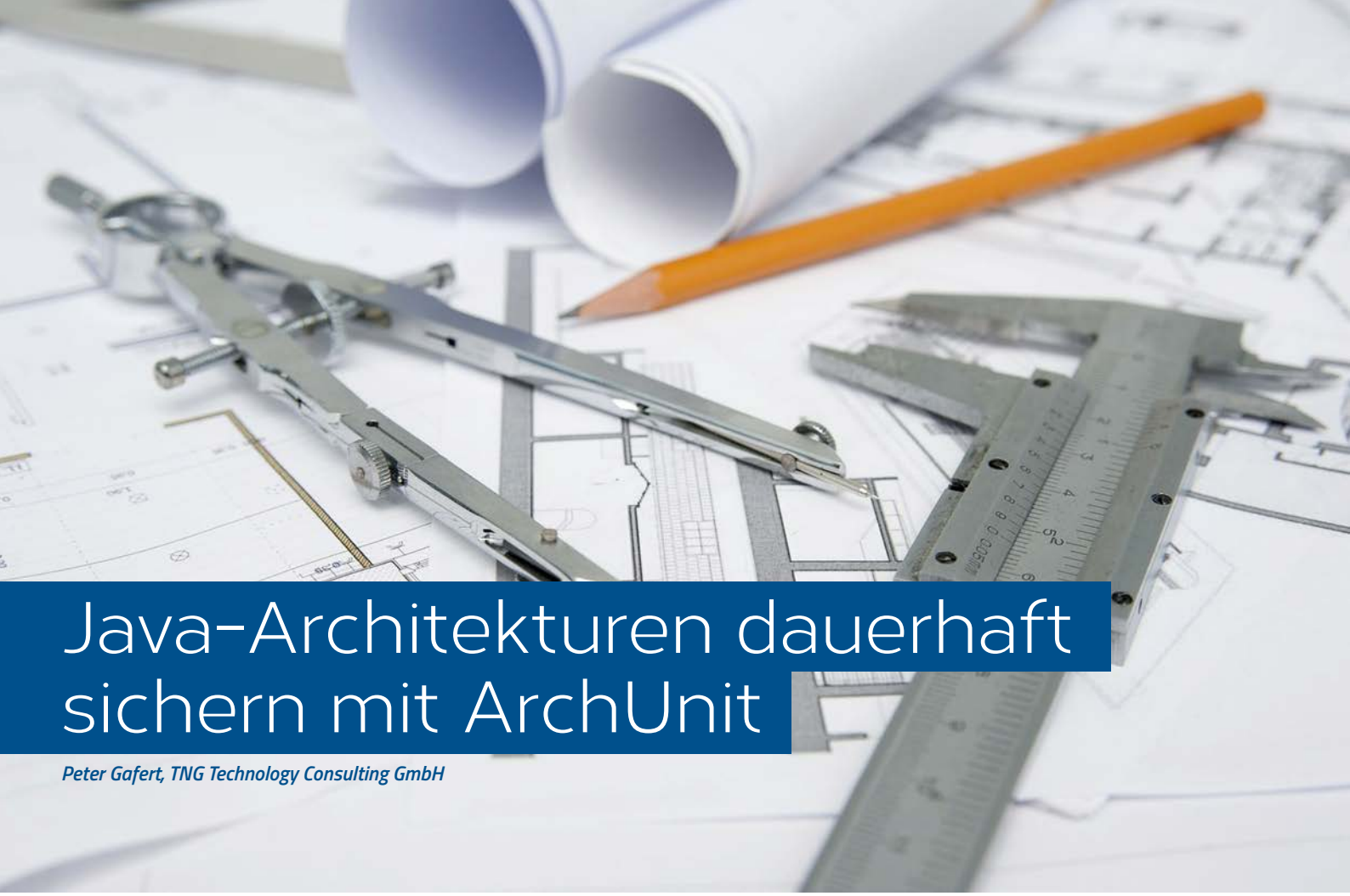

Java-Architekturen dauerhaft sichern mit ArchUnit

Peter Gafert, TNG Technology Consulting GmbH

Wenn man den Ist-Zustand gewachsener Software-Systeme betrachtet und sich von den Teams oder verantwortlichen Architekten deren Soll-Architektur erklären lässt, wird man häufig gravierende Abweichungen feststellen. Die Gründe sind verschiedener Natur; manchmal konnten die Konzepte nicht leisten, was benötigt wurde, manchmal wurden Features unter Zeitdruck entwickelt und manchmal waren neue Entwickler mit den Konzepten noch nicht ausreichend vertraut.

Bei einem großen System lassen sich Verletzungen der Architektur nicht mal eben nebenher reparieren. Der Verbleib der Verletzungen im Code verleitet Entwickler wiederum dazu, diese schlechten Muster an anderer Stelle zu kopieren, wodurch sich Architektur-Verletzungen auf Code-Basis schnell ausbreiten und eine Reparatur im Laufe der Zeit immer teurer machen.

Zudem ist der Verbleib der Verletzungen auf Dauer nicht kostenfrei. Der Preis, den man dafür zahlen muss, besteht aus mangelhafter Anpassbarkeit und Wartbarkeit. Abhängigkeiten, die unnötiger- und unerwarteterweise auftreten, lassen die tatsächlich benötigte Entwicklungszeit im Vergleich zu Schätzungen explodieren und führen zu neuen Bugs.

Eine Verletzung von Hierarchien und Mustern oder Neueinführung unnötiger Muster erschwert das Verständnis für jeden einzelnen Entwickler erheblich, was die Entwicklungsgeschwindigkeit weiter drückt. Außerdem entsteht ein Kommunikations-Overhead, wenn Entwickler versuchen, die Gründe für verletzte Muster und Hierarchien nachzuvollziehen.

Die Feature-Entwicklung hat über die letzten Jahrzehnte große Fortschritte gemacht und die Industrie hat begriffen, dass wir ohne ein Gerüst aus automatisierten Tests die Funktionalität der Software nicht kontinuierlich und effizient sicherstellen können. Die These dieses Artikels lautet, dass wir diese Erkenntnis, wenn möglich, auch auf die Architektur der Software anwenden sollten – also unsere Architektur-Regeln in einer programmatisch verständlichen Art festhalten und das System kontinuierlich dagegen validieren. Um solche Tests umzusetzen, sollte die Infrastruktur einige Features mitbringen:

- Sie muss möglichst viel Information über die Code-Basis des Projekts zur Verfügung stellen
- Regeln zu spezifizieren, muss einfach für Standardfälle und frei erweiterbar für Spezialfälle sein

- Die Installation und Integration in lokale und zentrale Testumgebungen muss so einfach wie möglich sein
- Die Funktionsweise muss klar, verständlich und wartbar sein, um eine flache Lernkurve zu gewährleisten

Der Artikel erläutert die Open-Source-Bibliothek ArchUnit und wie diese die genannten Kriterien erfüllt.

Aufbau von ArchUnit

Die Struktur von ArchUnit besteht aus mehreren Schichten, von denen ein Teil erläutert wird, beginnend mit der Core-Schicht. Bezüglich der automatisierten Sicherstellung der Architektur haben wir bei Java Glück, da sich ein Großteil der Struktur im statisch kompilierten Bytecode wiederfindet. Einige Eigenschaften lassen sich mit dem bekannten Java-Reflection-API abfragen und damit auch sicherstellen *(siehe Listing 1)*.

```
Method method = SomeController.class.
getDeclaredMethod("call");
assertThat(method.getAnnotation(UiAccess.class))
  .as("Controller dürfen keinen UI-Zugriff deklarieren")
  .isNull();
```

Listing 1

Sobald diese Eigenschaften allerdings nicht mehr lokal sind, wie zum Beispiel Abhängigkeiten zwischen zwei Klassen, bietet uns das Reflection-API diese Information nicht mehr an. Dennoch findet sich diese im Bytecode wieder; *Listing 2* dient als Beispiel. Wenn wir den Bytecode analysieren, lässt sich der Aufruf von „CallingClass" nach „OtherClass" klar nachvollziehen *(siehe Listing 3)*. Die Basis von ArchUnit orientiert sich nun an dem Java-Reflection-API, erweitert es jedoch um die fehlende Information. Das hat den Vorteil, dass dieses Core-API den meisten Entwicklern bereits in ihren Grundzügen vertraut ist.

```
class CallingClass {
  OtherClass other;

  void execute() {
     other.call();
  }
}
class OtherClass {
  void call() {
  }
}
```

Listing 2

Core-Objekte sind nach ihrem Reflection-Pendant mit einem zusätzlichen Java-Präfix benannt. So existieren zum Beispiel „JavaClass", „JavaMethod" und „JavaField", aber auch neue Konzepte wie „JavaMethodCall", die Methodenaufrufe repräsentieren. Grundsätzlich verhalten sich diese Objekte wie ein erweitertes Reflection-API. So bietet „JavaClass" zum Beispiel die Methode „getSimpleName()", die sich wie erwartet verhält. Auf der anderen Seite ist es auch mög-

```
javap -v CallingClass.class
...
  #2 = Fieldref   // CallingClass.other:LOtherClass;
  #3 = Methodref  // OtherClass.call:()V
...
void execute();
  descriptor: ()V
  flags:
  Code:
     stack=1, locals=1, args_size=1
        0: aload_0
        1: getfield      #2 // Field other:LOtherClass;
        4: invokevirtual #3  // Method OtherClass.call:()V
        7: return
     LineNumberTable:
        line 7: 0
        line 8: 7
...
```

Listing 3

```
// Der ClassFileImporter kann zahlreiche Quellen
// von Classpath bis File URL importieren
JavaClasses classes = new ClassFileImporter().importJar(new JarFile("/some/archive.jar"));

Set<JavaClass> services = new HashSet<>();
for (JavaClass clazz : classes) {
  // Wir filtern die Klassen, deren vollqualifizierter Name
  // den Paketinfix '.service.' enthält
  if (clazz.getName().contains(".service.")) {
     services.add(clazz);
  }
}

for (JavaClass service : services) {
  // Wir iterieren über alle Zugriffe, die von der Klasse ausgehen
  for (JavaAccess<?> access : service.getAccessesFromSelf()) {
     String targetName = access.getTargetOwner().getName();

     // Fehlschlag, falls Zugriff auf ein Ziel mit ".controller." im Namen
     if (targetName.contains(".controller.")) {
        String message = String.format(
           "Service %s accesses Controller %s in line %d",
           service.getName(), targetName, access.getLineNumber());
        Assert.fail(message);
     }
  }
}
```

Listing 4

```
ArchRule rule =
  classes().that().resideInAPackage("..service..")
     .should().onlyBeAccessed().byAnyPackage("..controller..", "..service..");
```

Listing 5

```
JavaClasses classes = new ClassFileImporter().importPackage("com.myapp");
ArchRule rule = // siehe oben
rule.check(classes);
```

Listing 6

```
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] -
Rule 'classes that reside in a package ,..service..' should only be accessed by any package ['..controller..',
'..service..']' was violated:
Method <com.myapp.dao.EvilDao.callService()> calls method <com.myapp.service.SomeService.doSomething()> in (EvilDao.
java:14)
```

Listing 7

```
ArchRule layeredArchitecture = layeredArchitecture()
  .layer("Controllers").definedBy("..controller..")
  .layer("Services").definedBy("..service..")
  .layer("Persistence").definedBy("..persistence..")
  .whereLayer("Controllers").mayNotBeAccessedByAnyLayer()
  .whereLayer("Services").mayOnlyBeAccessedByLayers("Controllers")
  .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Services");
layeredArchitecture.check(classes);
```

Listing 8

```
ArchRule servicesShouldBeFreeOfCycles =
  slices().matching("com.myapp.(**).service..").should().beFreeOfCycles();
servicesShouldBeFreeOfCycles.check(classes);
```

Listing 9

lich, Dinge wie „javaClass.getAccessesToSelf()" abzufragen, was einem wiederum Zugriffe im analysierten Bytecode auf die jeweilige Klasse liefert.

Der Core-Layer beinhaltet ebenfalls den „ClassFileImporter", um Bytecode in eine programmatisch verständliche Form zu bringen. *Listing 4* zeigt ein Beispiel für einen möglichen Test.

Lang-Schicht

Eine reine Verwendung der Core-Klassen für Architektur-Tests ist zwar möglich, allerdings fehlt hier die Ausdrucksstärke. Um diese zu gewährleisten, bietet ArchUnit eine höhere Abstraktionsschicht, die ein Fluent-API und damit IDE-Unterstützung liefert. Das API baut auf den im letzten Absatz beschriebenen Core-Klassen auf und führt einige höhere Konzepte ein:

- *ArchRule*
  Architekturelle Konzepte sind als Regeln festgehalten
- *DescribedPredicate*
  Ein Prädikat zur Auswahl relevanter Klassen
- *ArchCondition*
  Eine Bedingung, die ausgewählte Klassen erfüllen müssen

Durch eine Kombination dieser Elemente lassen sich statische Architektur-Constraints nun beschreiben und automatisch sicherstellen, denn prinzipiell sind die meisten Regeln folgendermaßen aufgebaut: „classes that $PREDICATE should $CONDITION". Im Code könnte sich diese Idee zum Beispiel in folgender Form wiederfinden *(siehe Listing 5)*.

In diesem Beispiel lautet das Prädikat „resideInAPackage(„..service..")", das die zu untersuchenden Klassen auf solche einschränkt, die ein Paket „service" im vollqualifizierten Klassennamen haben. Die Syntax mit den „.." orientiert sich hier an „AspectJ". Die Bedingung an diese Klassen lautet wiederum „onlyBeAccessed().byAnyPackage(„..controller..", „..service..")", die ausgewählten Klassen dürfen also nur von Klassen aufgerufen werden, die im vollqualifizierten Klassennamen ein Paket „controller" oder „service" enthalten. Hat man eine Regel in der obigen Form definiert, lässt sie sich einfach gegen importierte „JavaClasses" auswerten *(siehe Listing 6)*. Bei einem Fehlschlag erhält man eine detaillierte Fehlermeldung inklusive Zeilennummer *(siehe Listing 7)*.

Library-Schicht

Oberhalb von Lang liegt die Library-Schicht. Prinzipiell ist hier eine wachsende Sammlung von typischen Standardfällen angesiedelt. Ein Beispiel ist eine vorgefertigte Sicherstellung einer Schichten-Architektur, die sich in bekannter Weise einbinden lässt *(siehe Listing 8)*. Eine weiterer typischer Anwendungsfall ist die Prüfung auf Zyklen zwischen gewissen fachlichen oder technischen Schnitten *(siehe Listing 9)*.

Hier würden Klassen nach den geklammerten Paketteilen in Schnitte gruppiert – „(**)" steht für beliebig viele Pakete – und diese dann auf Zyklenfreiheit geprüft. Beispielsweise würde die Klasse „com.myapp.order.service.OrderService" im Schnitt „order" landen, wohingegen die Klasse „com.myapp.customer.relations.service.SomeThing" dem Schnitt „customer.relations" zugeordnet werden würde. Greift nun eine beliebige Klasse aus dem Schnitt „order" auf eine Klasse aus dem Schnitt „customer.relations" zu und zudem eine beliebige Klasse aus dem Schnitt „customer.relations" auf den Schnitt „order", so schlägt die Regel fehl und meldet die Details des unerwünschten Zyklus.

JUnit

Momentan bietet ArchUnit optimierte Unterstützung für JUnit 4. Hier existiert ein spezieller Runner, der die importierten Klassen nach URLs cacht, sodass Klassen für mehrere Tests nicht jedes Mal erneut importiert werden müssen. Auch das Auswerten der Regeln muss man in diesem Fall nicht selbst durchführen. Es reicht, „ArchRule"-Felder zu deklarieren *(siehe Listing 10)*.

Eigene Konzepte mit ArchUnit definieren

ArchUnit bringt viele vorgefertigte Regeln mit, beispielsweise das vorgestellte Fluent-API, das IDE-Unterstützung für typische Anwendungsfälle wie Zugriffe von gewissen Klassen auf gewisse Klassen ermöglicht. Grundsätzlich sind diese Regeln immer in der Form „Klassen [Einschränkung, z.B. im Paket service'] sollen [Bedingung]" definiert. Das Fluent-API nutzt dabei im Zentrum ein generisches API der Form „classes.that(predicate).should(condition)". Dabei ist „predicate" vom Typ „DescribedPredicate", also ein Prädikat, das für die Regel relevante Klassen auswählt und eine Beschreibung für die Erstellung des Regeltexts mitliefert. Das Argument „condition" ist vom Typ „ArchCondition" und erlaubt die

```
@RunWith(ArchUnitRunner.class)
@AnalyzeClasses(packages = "com.myapp")
public class MyArchitectureTest {

  @ArchTest
  public static final ArchRule firstRule =
     classes().that().areAnnotatedWith(PayLoad.class)
        .should().beAssignableTo(Serializable.class);

  @ArchTest
  public static final ArchRule secondRule = // ...

}
```

Listing 10

```
DescribedPredicate<JavaClass> resideInAPackageService
= // define predicate...
ArchCondition<JavaClass> onlyBeAccessedByAnyPackage-
ControllerOrService = // define condition...

classes()
  .that(resideInAPackageService)
  .should(onlyBeAccessedByAnyPackageControllerOrServi
ce);
```

Listing 11

```
ClassesTransformer<Slice> slices = // specify how to
transform classes to slices
ArchCondition<Slice> beFreeOfCycles = // check for
cycles between slices
ArchRule slicesRule = all(slices).
should(beFreeOfCycles);
```

Listing 12

```
@ArchTest
public static final ArchRule customConcepts =
  all(businessModules()).that(dealWithOrders()).should(beIndependentOfPayment());

static ClassesTransformer<BusinessModule> businessModules() {
  return new AbstractClassesTransformer<BusinessModule>("business modules") {
     @Override
     public Iterable<BusinessModule> doTransform(JavaClasses classes) {
        // how we map classes to business modules
     }
  };
}

static DescribedPredicate<BusinessModule> dealWithOrders() {
  return new DescribedPredicate<BusinessModule>("deal with orders") {
     @Override
     public boolean apply(BusinessModule module) {
        // return true, if a business module deals with orders
     }
  };
}

static ArchCondition<BusinessModule> beIndependentOfPayment() {
  return new ArchCondition<BusinessModule>("be independent of payment") {
     @Override
     public void check(BusinessModule module, ConditionEvents events) {
        // check if the actual business module is independent of payment
     }
  };
}
```

Listing 13

Implementierung einer Bedingung an Klassen. Die vorgestellte Regel für Paketzugriffe ist lediglich eine spezielle Verwendung dieses API *(siehe Listing 11)*.

Auf diese Art und Weise lassen sich beliebige eigene Predicates und Conditions erstellen, um spezifische Architektur-Konzepte des bestehenden Projekts festzuhalten. Tatsächlich geht ArchUnit aber noch einen Schritt weiter, denn die oben vorgestellte Regel, die sich auf Zyklenfreiheit von „Slices" bezieht, basiert ebenfalls auf einem generischen API, das man für seine Architektur-Konzepte selbst verwenden kann. Die grundsätzliche Regelform bleibt auch hier erhalten, jedoch sind die Objekte, über die man sprechen will, eventuell keine Klassen. Grundsätzlich könnte man die obige Regel für Zyklen auch in folgender Form schreiben *(siehe Listing 12)*. Man kann daher genau an das eigene Projekt angepasste Konzepte einführen *(siehe Listing 13)*.

Der Regeltext wird, sofern nicht explizit überschrieben, jeweils dynamisch aus den spezifizierten Teilbeschreibungen gebildet, in diesem Beispiel wäre der berechnete Text der ausgeführten Regel also bereits „business modules that deal with orders should be independent of payment".

ArchUnit im Projektalltag

Beginnt man, Regeln mit einem Tool wie ArchUnit zu schreiben, fallen einem ganz natürlich immer weitere Anwendungsfälle auf. Beispielsweise lässt sich die fehlerhafte Nutzung von 3rd-Party-Libraries einfach für die Zukunft vermeiden, man möchte Fehler ja nur einmal machen, oder gewisse Code-Smells lassen sich aus dem Projekt verbannen, indem man gezielt gegen diese Muster testet.

Der größte Wert entsteht aber durch das Lebendighalten der Architektur. Während sich der Code-Stand unter normalen Umständen stillschweigend von der gewünschten Soll-Architektur und den gewünschten Konzepten wegentwickelt, schlagen beim Einsatz von ArchUnit tatsächlich explizit Tests im CI-Server an, beziehungsweise bereits beim lokalen Testlauf vor dem Commit. Durch diese Testfehlschläge muss man sich wirklich mit der Abweichung auseinandersetzen. Hier kann es entweder sein, dass man die bestehenden Konzepte aus Unaufmerksamkeit verletzt hat. Es ist aber durchaus auch möglich, dass die gewünschten Konzepte für den Anwendungsfall nicht mehr ausreichen. In einem solchen Fall möchte man sich allerdings explizit überlegen, wie die bestehenden Konzepte erweitert werden sollen, um derartige Anwendungsfälle abdecken zu können. Ansonsten entstehen willkürliche Muster, die dann an anderer Stelle ohne hinreichenden Grund einfach übernommen werden und sich so im System ausbreiten.

Der Erhalt einer konsistenten und sauberen Struktur des Systems ist kein Selbstzweck, sondern eine Grundvoraussetzung, um das System ab einer gewissen Größe wartbar und die Entwicklungsgeschwindigkeit konstant zu halten. Die Kosten für den Erhalt dieser Struktur wiederum sind desto geringer, je schneller man Feedback bei Verletzungen bekommt – das macht die kontinuierliche Prüfung so essenziell.

Fazit

ArchUnit *(siehe „http://www.archunit.org")* ist eine Open-Source-Bibliothek, die viele Möglichkeiten mitbringt, um architekturelle Konzepte dauerhaft sicherzustellen und lebendig zu dokumentieren. Sie ist entwicklerfreundlich und eignet sich besonders gut für größere agile Projekte, in denen Architektur-Verantwortung oft verteilt ist und viele Teams parallel an einer sich stetig weiterentwickelnden Code-Basis arbeiten.

ArchUnit benötigt keine weitere Infrastruktur und kann als Library in jedes beliebige Java-Projekt eingebunden werden, um Architektur auf Unit-Test-Ebene sicherzustellen. Zudem bietet ArchUnit ein mächtiges API, mit dem nicht nur auf viele Eigenschaften im Java-Byte-Code reagiert werden kann, sondern auch auf abstrakter Ebene Architektur-Konzepte dokumentiert und automatisiert getestet werden können.

Abschließend sei erwähnt, dass sich automatisierte statische Architekturtests zur Architektur ähnlich wie Test-Coverage zu Testqualität verhalten. So wie 100 Prozent Test-Coverage in keiner Weise garantiert, dass ein System gut getestet ist, so garantiert ein erfülltes System von statischen Architektur-Regeln nicht, dass die Architektur gut ist und Ziele wie etwa geringe Kopplung erfüllt sind. Schließlich kann man beispielsweise sämtliche Aufrufe durch Nutzung des Reflection-API ersetzen, und man wird alle statischen Regeln bezüglich Abhängigkeiten erfüllen. Allerdings gilt im Umkehrschluss entsprechend der Tatsache, dass 0 Prozent Test-Coverage ein Garant für ein schlecht getestetes System ist, dass eine Verletzung der Architektur auf statischer Ebene mit Sicherheit auf ein System hinweist, das die gewollten Architektur-Ziele verfehlt. Insofern kann man die statische Analyse als grundsätzlichen Sanity-Check verstehen, der auf dieser Ebene einen großen Wert mit sich bringt – insbesondere für große Systeme und inhomogene Entwicklungsteams.

Peter Gafert
peter.gafert@tngtech.com

Peter Gafert ist Senior Consultant bei der TNG Technology Consulting GmbH und beschäftigt sich im Projektalltag viel mit Software-Architektur. Um den täglichen Umgang mit Architektur zu verbessern, entwickelt er nebenher die Open-Source-Library ArchUnit.