Default-Methoden in Java-Interfaces?

Endlich viel erben?

Phillip Ghadir, Oliver Tigges

Im Fahrwasser der Lambda-Funktionen werden in Java 8 Default-Methoden eingeführt, um Interfaces aus dem JDK abwärtskompatibel um neue Methoden zu erweitern. Hier schließen wir an den Artikel von Christian Robert vom letzten Jahr an. Wir klären, bis zu welchem Grad Mehrfacherbung in Java einzieht und wie wir die Möglichkeiten und das Risiko für bestehende Systeme einschätzen.

Java8.getShortInfo()

Nach mehrmaligen Verschiebungen ist das Release von Java 8 nun für März 2014 [Rel] angekündigt. Es wird unter anderem folgende Neuerungen bringen:

- Lambdas,
- Zugriff auf Parameternamen zur Laufzeit per Reflection,
- Mengenoperationen für Collections,
- Reduzierter Speicherbedarf der Kern-Bibliotheken.

Uns interessiert hier vor allem der Mechanismus, mit dem die Standard-APIs des JDK erweitert wurden, um Lambda-Funktionen zu unterstützen: die Default-Methoden. Es ist bemerkenswert, dass sie gar nicht in den JDK 8-Features aufgeführt werden (siehe [Ftr]).

Alte Schnittstellen – neue Methoden

Unter dem unscheinbaren Punkt „Mengenoperationen für Collections“ verbirgt sich eine Menge interessanter Implikationen. Beispielsweise wurden die zentralen Schnittstellen der Collection-API um Methoden erweitert, die Lambdas als Argument verstehen. So erhält zum Beispiel das Interface **`java.util.Iterable`** – das alle Collection-Klassen implementieren – in Java 8 die neue Methode

```
void forEach(Consumer<? super T> action);
```

Das Hinzufügen neuer Methoden in eine Schnittstelle bricht erst einmal die implementierenden Klassen. Eigentlich müssten dann alle Klassen angepasst werden, die diese Schnittstelle direkt oder indirekt implementieren: die Klassen des JDK, die Klassen aus Bibliotheken von Drittanbietern sowie unsere eigenen Klassen, die von **`Iterable`** oder **`Collection`** ableiten. Davon wären also potenziell alle Java-Anwendungen betroffen, die auf Java 8 portiert würden!

Default-Kompatibilität

Um das zu verhindern und die Erweiterung abwärtskompatibel zu gestalten, wird in Java 8 parallel zu den Lambda-Funktionen ein weiteres Konstrukt eingeführt: die Default-Methode (teilweise auch „Defender Methods“ oder „Virtual Extension Methods” genannt).

Durch Voranstellen des Schlüsselworts **`default`** vor die Methodendeklaration in einem Interface wird angezeigt, dass nun eine Definition der Methode erfolgt. An die Signatur wird dann einfach wie gewohnt der Methodenrumpf angefügt. Diese Default-Implementierung der Methode greift immer dann, wenn nicht eine andere Implementierung greift.

Traits in Scala

Traits ermöglichen, ähnlich wie abstrakte Klassen, Eigenschaften und Methoden zusammenzufassen und wiederzuverwenden. Für einen Java-Entwickler ist ein Trait eine Mischung aus Interface und abstrakter Klasse: Methoden können deklariert und auch direkt implementiert werden. Darüber hinaus können Traits auch Instanz-Attribute definieren und damit ihren eigenen Zustand verwalten.

Eine Klasse kann in Scala von mehreren Traits erben. Damit bilden Traits eine Art Mittelweg zwischen vollständiger Mehrfacherbung wie in C++ und einfachem Erben in Kombination mit dem Realisieren von Interfaces wie in Java.

Die mit Java 8 eingeführten Default-Methoden sind ein Schritt von Java in Richtung Traits. Auch das explizite Auflösen von Konflikten bei Default-Methoden erinnert an das Vorgehen in Scala – wie das folgende Beispiel zeigt.

```
trait Eingabe { def einschalten = println(“Bereit zur Eingabe”) }
trait Ausgabe { def einschalten = println(“Bereit zur Ausgabe”) }

class Touchscreen extends Eingabe with Ausgabe {
  override def einschalten = super[Eingabe].einschalten
}
```

Es werden zwei Traits mit jeweils einer Methode **`einschalten`** definiert. In der abgeleiteten Klasse wird der Namenskonflikt durch eine überschreibende Methode aufgelöst, die explizit an die gewünschte Trait-Implementierung delegiert.

Zum Beispiel sieht für die oben genannte Methode **`forEach( ... )`** in **`Iterable`** die Standardimplementierung ungefähr so aus, auch wenn sie je nach JDK ein wenig davon abweichen kann

```
public interface Iterable<T> {
  //...
  default void forEach(Consumer<? super T> action) {
    for (T t : this)
      action.accept(t);
  }
  //...
}
```

Die Default-Methoden gewähren uns damit Abwärtskompatibilität und erlauben das Definieren von Implementierungen für neue Methoden in Schnittstellen. Damit lässt sich das oben erwähnte Anpassen der Klassen vermeiden, die solche erweiterten Schnittstellen implementieren.

In Java-Interfaces Default-Methoden zu definieren, erinnert uns an die Traits aus der Programmiersprache Scala (s. Kasten „Traits in Scala“).

Intention der Default-Methoden

Beim Design der Default-Methoden stand die „Evolution von Interfaces“ im Vordergrund. Die Expertengruppe [LambdaDev] wollte Komplexität vermeiden. Bislang konnte die Komplexität von Mehrfacherbung aus Java herausgehalten werden. Auch mit Java 8 gibt es keine Einführung von Traits, wie wir sie aus Scala kennen.

Die Expertengruppe hat drei einfache und klare Regeln aufgestellt, nach denen entschieden werden kann, welche Methodenimplementierung einer deklarierten Methode in einer Hierarchie von Klassen und Interfaces verwendet wird. Die Regeln lauten:

1.) Klassen gewinnen immer über Interfaces. Sobald eine Methode von einer Klasse – egal wo sie in der Hierarchie steht – implementiert ist, werden alle Default-Methoden ignoriert.
2.) Spezifische Interfaces gewinnen über weniger spezifische, also zum Beispiel `List` über `Collection`.
3.) Sobald durch die Regeln 1.) und 2.) nicht klar definiert ist, welche Methodenimplementierung gewinnt, muss der Entwickler den Konflikt selbst in der implementierenden Klasse auflösen.

Einschränkungen

Mit einer Default-Implementierung lassen sich gemäß der beschriebenen Regeln keine Methoden überschreiben, die bereits von einer Klasse implementiert sind. Einer der ersten kreativen Anwendungsfälle wäre aus unserer Sicht die Prüfung auf Gleichheit oder String-Ausgabe für einen Stereotyp von Klassen – wie zum Beispiel Entitäten – gewesen.

Allerdings sind `equals()` und `toString()` bereits in `java.lang.Object` definiert, sodass diese Idee direkt nur mit eigenen Methoden (und darauf aufbauenden Frameworks oder Bibliotheken) oder dem üblichen Weg der gewöhnlichen Vererbung durch eine Superklasse funktioniert.

Diese Beschränkung verhindert das Überschreiben einer von einer Superklasse geerbten Funktionalität durch eine neue Default-Methode in einer Schnittstelle. Das Ziel der Abwärtskompatibilität wird dadurch gut unterstützt und jede unliebsame Überraschung vermieden.

Es ist auch mit Java 8 nicht möglich, in einem Interface Attribute zu deklarieren sowie Konstruktoren, Initialisierungsblöcke oder `finalize()` zu definieren. Diese Konstrukte werden weiterhin nur in Klassen verwendet.

Auswirkung der Default-Methoden

Wir wollen nun anhand einiger Szenarien die Auswirkungen oder gar Überraschungen bei der Verwendung von Default-Methoden demonstrieren. Dazu stellt Abbildung 1 das Modell dar, anhand dessen wir einige Thesen überprüfen wollen.

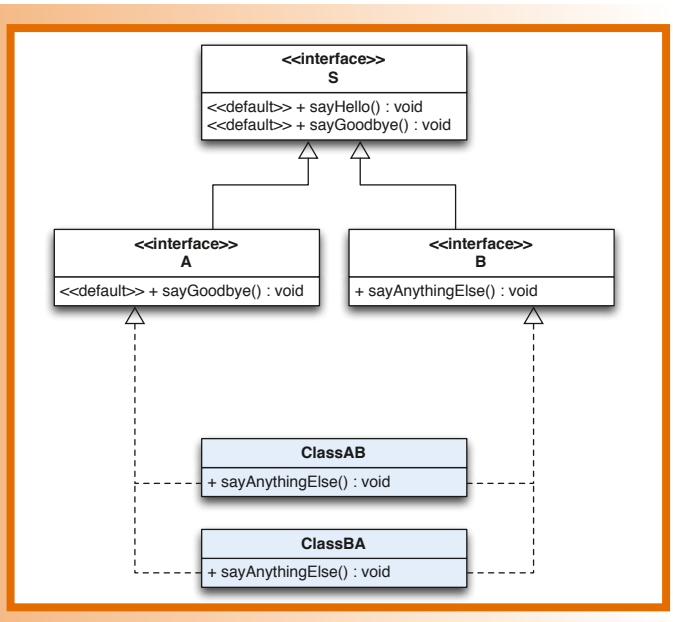

Abb. 1: Die statische Struktur unseres Test-Setups

Die in der Abbildung dargestellte Schnittstelle `S` definiert die beiden Default-Methoden `sayHello()` und `sayGoodbye()`. Die hiervon abgeleitete Schnittstelle `A` implementiert ebenfalls `sayGoodbye()` als Default-Methode. Alle Implementierungen geben einfach nur den Namen des definierenden Interfaces sowie den Methodennamen aus.

Zusätzlich haben wir die Schnittstelle `B` definiert, die ebenfalls von dem Super-Interface `S` erbt, aber `sayGoodbye ()` nicht redefiniert, sondern zusätzlich eine weitere Methode deklariert.

Das Setup für die Szenarien enthält noch die Klassen `ClassAB` und `ClassBA`. Die Namen symbolisieren, in welcher Reihenfolge sie welche der Schnittstellen implementieren. `ClassAB` implementiert die von Interface `B` deklarierte Methode und sonst nichts. Beim Aufruf von

```
new ClassAB().sayHello();
```

wird problemlos die Default-Implementierung aus dem Super-Interface `S` aufgerufen. Bei einem Aufruf von

```
new ClassAB.sayGoodbye();
```

wird hingegen die Default-Implementierung aus dem Interface `A` aufgerufen, da diese die Implementierung aus dem Super-Interface überschreibt.

Die Klassen `ClassAB` und `ClassBA` realisieren die Schnittstellenabhängigkeiten zu den Interfaces `A` und `B` – aber in umgekehrter Reihenfolge. Das Ergebnis ist dennoch in beiden Fällen identisch, es wird die `sayGoodbye()`-Methode aus `A` ausgeführt. Die Auswahl der Default-Methode hängt also nicht von der Reihenfolge der `implements`-Deklaration ab.

Das Diamond-Problem

Durch die Einführung der Default-Methoden haben wir nun auch in Java die Möglichkeit, mehrere Schnittstellen zu realisieren, die alle eine eigene Default-Implementierung mitbringen.

Abbildung 2 stellt am Beispiel eines Touchscreens das sogenannte Diamond-Problem vor. Das Problem tritt auf, wenn zwei Klassen von derselben Superklasse erben und von diesen zwei Klassen wiederum eine weitere Klasse erbt. Beispielsweise erben die Klassen `Eingabegerät` und `Ausgabegerät` beide von der Klasse `Gerät`, während `Touchscreen` die Attribute und Methoden von `Eingabegerät` und `Ausgabegerät` erhält und in diesem Fall zweimal von der Klasse `Gerät` erbt.

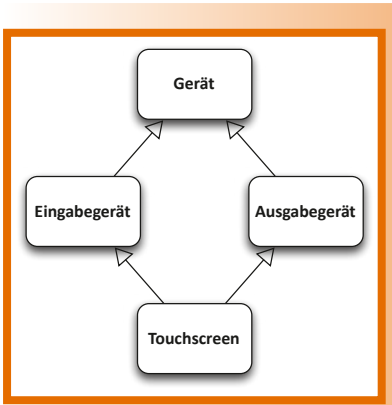

Abb. 2: Ein hinkendes fachliches Beispiel, das die diamantförmigen Erbungsbeziehungen von Gerät und Touchscreen mit den Zwischenstufen Ein- und Ausgabegerät zeigt

Szenarios mit dem Touchscreen-Beispiel

Das klassische Java-Interface zu dem oben genannten Beispiel sieht in Java so aus

```
interface Eingabegeraet { void einschalten(); }
```

Wenn eine Klasse dieses Interface implementiert

```
class Touchscreen implements Eingabegeraet {}
```

wird der Compiler sich beschweren, dass die deklarierte Methode `einschalten()` von der Klasse `Touchscreen` nicht implementiert worden ist. Sofern es ein sinnvolles Standardverhalten für den Einschaltvorgang gibt, kann dies in Java 8 jetzt prinzipiell im Interface definiert werden

```
interface Eingabegeraet { default void einschalten() { ... }; }
```

Die Klasse `Touchscreen` lässt sich jetzt kompilieren. Nun fügen wir das zweite Interface für das Ausgabegerät hinzu und deklarieren ebenfalls eine Methode `einschalten()`, aber ohne Default-Implementierung

```
interface Ausgabegeraet { void einschalten(); }
```

Die Klasse `Touchscreen` lassen wir nun beide Interfaces implementieren

```
class Touchscreen implements Eingabegeraet, Ausgabegeraet {}
```

Was meldet der Compiler jetzt? Obwohl `einschalten()` im Eingabegerät implementiert ist, wird sich der Compiler beschweren und die Klasse nicht übersetzen, weil die Methode `Ausgabegeraet.einschalten()` nicht implementiert ist. Eine Default-Methode in einem Interface `A` kann also nicht als Implementierung einer Methode mit gleicher Signatur in Interface `B` verwendet werden. Passen wir also das `Ausgabegeraet` um eine Default-Implementierung an

```
interface Ausgabegeraet { default void einschalten() { ... }; }
```

Der Compiler meldet jetzt den nächsten Konflikt. Gemäß der oben genannten Regeln für das Auffinden der richtigen Implementierung ist unklar, ob bei einem Aufruf von `Touchscreen.einschalten()` die Implementierung von `Eingabegeraet` oder `Ausgabegeraet` verwendet werden soll. Beide Implementierungen stehen in der Klassenhierarchie auf derselben Ebene. So überschreibt erst einmal keine der gleichnamigen Methoden die andere. Java verzichtet darauf, den Namenskonflikt implizit zu lösen – zum Beispiel durch die Reihenfolge, in der eine Klasse die Interfaces implementiert. Das hat uns auch der Versuch mit den Klassen `ClassAB` und `ClassBA` bestätigt.

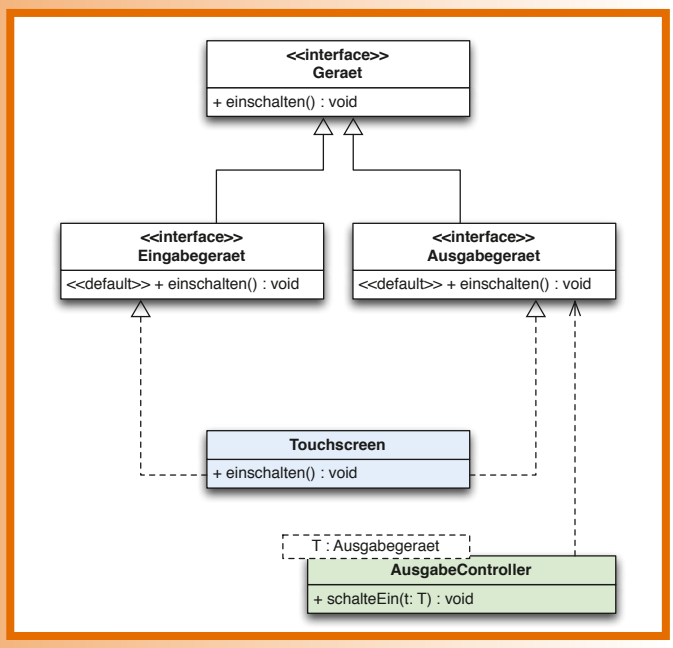

Abb. 3: Auswirkungen von Default-Methoden im Zusammenhang mit Generics

Es greift also Regel 3.). Der Entwickler muss solch einen Konflikt explizit auflösen

```
class Touchscreen implements Eingabegeraet, Ausgabegeraet {
  public void einschalten() {
    Ausgabegeraet.super.einschalten();
  }
}
```

Die einzelnen Default-Methoden können über den Namen des Interfaces, gefolgt vom Schlüsselwort `super` direkt adressiert werden. Der Konflikt ist behoben und der Compiler kann die Klasse jetzt übersetzen. Das Verhalten und auch die Syntax erinnern hier an die Traits aus Scala.

Werkzeuge für Diamanten

Wir möchten noch einmal das Beispiel mit dem Diamond-Problem aufnehmen und es um eine interessante Facette erweitern. In der in Abbildung 3 dargestellten Konstellation implementieren die Interfaces `Eingabegeraet` und `Ausgabegeraet` beide eine Default-Implementierung von `einschalten()`, in der sie wieder einfach ihren Namen und die aufgerufene Methode ausgeben.

In der Klasse `Touchscreen` wird die Mehrdeutigkeit aufgelöst, indem dort definiert wird, dass stets an die Implementierung von `Eingabegeraet` delegiert wird

```
@Override
public void einschalten() {
  Eingabegeraet.super.einschalten();
}
```

Der `AusgabeController` erwartet in der einzig definierten Methode `schalteEin` einen Parameter, der sich an den in `Ausgabegeraet` definierten Kontrakt hält. Nun wollen wir der parametrisierbaren Hilfsklasse `AusgabeController` eine `Touchscreen`-Instanz vorwerfen, deren Methode `einschalten()` sich wie ein Eingabegerät verhält.

Zur Vereinfachung rufen wir einfach die Methode `schalteEin()` auf. `AusgabeController<T extends Ausgabegeraet>.schalteEin(T t)` ist wie folgt definiert

```
public void schalteEin(T t) {
  t.einschalten();
}
```

Die spannende Frage ist nun, was passiert? Gemäß der gelernten Regeln sollte die Implementierung von `Eingabegeraet` aufgerufen werden, weil die Implementierung der `Touchscreen`-Instanz ja genau an diese delegiert. Aber die Generic-Deklaration verlangt eine Instanz von `Ausgabegeraet`, das unabhängig von `Eingabegeraet` ist. Die Lösung verrät uns der Aufruf

```
public static void main(String[] args) {
  AusgabeController<Touchscreen> controller =
    new AusgabeController<>();
  controller.schalteEin( new Touchscreen() );
}
```

Die Ausgabe ist:

```
>>>> Eingabegeraet.einschalten() was called!
```

Also: Der im Generic erwartete Typ wird zwar zur Compile-Zeit herangezogen, um die Typkompatibilität zu prüfen, aber der Compiler kann nicht sicherstellen, dass der fachliche Kontrakt von `Ausgabegeraet` eingehalten wird.

Fazit

Default-Methoden in Interfaces zu definieren, bringt Java 8 ein paar Vorzüge. Schnittstellen können nun um Methoden – samt Default-Implementierung – erweitert werden, sodass bestehende Klassen, die bereits die Schnittstellen realisieren, nicht angepasst werden müssen.

Wir haben gesehen, dass die Anpassungen sehr zurückhaltend eingepflegt wurden und deshalb viele der mit Mehrfacherbung assoziierten Probleme nicht auftreten können. Für Klassen gibt es nur einfaches Erben. Schnittstellen können weiterhin weder Attribute noch Konstruktoren oder Finalizer definieren. Die Mehrfacherbung ist in Java 8 also sehr begrenzt.

Insgesamt erinnern die Default-Methoden am ehesten an Scala-Traits. In unseren Gehversuchen haben wir keine echten Überraschungen erlebt. Das könnte sich natürlich ändern, sobald man wirklich anfängt, dieses Feature zu nutzen.

Wer sich echte Mehrfacherbung durch die Default-Methoden erhofft hat, wird enttäuscht sein. Wer sie hingegen befürchtet hat, wird vermutlich beruhigt sein.

Aus unserer Sicht ist das Feature Default-Methode keines, das wir persönlich gern in der Anwendungsentwicklung sehen würden. Sein Platz ist im Schatten, in Framework-Implementierungen. Aber vielleicht werden noch ein paar tolle Anwendungsfälle für die Default-Methoden entdeckt.

Bis dahin sind sie vor allem für eine Sache wirklich sinnvoll: Sie ermöglichen uns – wie in der Collection-API gezeigt – bestehende Interfaces zu erweitern, bei denen wir uns das bisher nicht getraut haben, weil wir um die Abwärtskompatibilität gefürchtet haben.

Mehrfacherbung

Wir sprechen von Mehrfacherbung – statt Mehrfach*ver*erbung, weil im Fokus die Klasse steht, die erbt. Wenn eine Klasse zu mehr als einer anderen Klasse eine erbt-von-Beziehung deklarieren kann, spricht man von Mehrfacherbung.

In [Gos95] hat James Gosling erklärt, warum es in Java keine Mehrfacherbung geben soll. Die Designer von Java kamen zu einem ähnlichen Schluss wie später unter anderem auch die Designer von C#: Die durch Mehrfacherbung eingeführte Komplexität übersteigt den zu erwartenden Nutzen.

Echte Mehrfacherbung erlaubt einer Klasse von mehreren Klassen direkt zu erben und damit alle Eigenschaften und Methoden der Superklassen zu erben.

Die Frage nach instanceof wird für eine Instanz für jede Klasse innerhalb des Vererbungsgraphen mit ja beantwortet. Beispielsweise kann ein `Touchscreen` von `Eingabegerät` und `Ausgabegerät` erben, um die jeweiligen Eigenschaften und Funktionen von beiden Basisklassen zu erhalten.

Allerdings birgt Mehrfacherbung auch ein paar Risiken und Probleme.

- *Hohe Komplexität:* Auch wenn die einzelnen Klassen in einer tiefen Klassenhierarchie alle überschaubar sind, kann einen die Komplexität des Zustandsraums und die Menge der Methoden in der abgeleiteten Klasse erschlagen. Das Problem von „Monsterklassen" kann zwar auch bei einfacher Erbung auftreten, bei Mehrfacherbung ist die Gefahr insbesondere aufgrund möglicher Widersprüche größer.
- *Unklar, wie Polymorphismus:* Wenn eine Klasse direkt von mehreren Superklassen erbt, die alle eine Methode mit derselben Signatur definieren, ist es sowohl für den Compiler als auch für den Entwickler schwierig, die Implementierung zu identifizieren, die letztendlich ausgeführt wird. Der Polymorphismus ist bei Mehrfacherbung und mehreren Klassenhierarchien noch einmal komplizierter und es muss sehr genau definiert werden, wie die jeweils gültigen Implementierungen gefunden werden. In C++ sind hierfür zum Beispiel Konzepte wie „Dominanz" ersonnen worden [MemberLookup].
- *Diamond-Problem:* Bei mehreren Superklassen, die sich wiederum eine gemeinsame Superklasse teilen, muss definiert werden, wie Verschattungen aufgelöst werden, wenn gleichnamige Attribute oder Methoden parallel in Superklassen definiert werden. Bei Attributen kommt die Frage der Initialisierungsreihenfolge und Vereinigung bzw. Isolation von Speicherbereichen ins Spiel. Sofern die Klassen Attribute verschatten – sich also Zustand teilen –, ist die Reihenfolge der Konstruktoraufrufe wichtig.

Literatur und Links

[Ftr] JDK 8, Features, `http://openjdk.java.net/projects/jdk8/features`

[Gos95] J. Gosling, Java: an Overview, 1995, `http://www.cs.dartmouth.edu/~mckeeman/cs118/references/OriginalJavaWhitepaper.pdf`

[LambdaDev] B. Goetz, Allow default methods to override Object's methods, 2013, `http://mail.openjdk.java.net/pipermail/lambda-dev/2013-March/008435.html`

[LambdaTutorial] Lambda Expressions, Beta Draft 2013-10-15, The Java™ Tutorials, `http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`

[MemberLookup] G. Ramalingam, A member lookup algorithm for {C}++, in: Proc. of the SIGPLAN '97 Conference on Programming Language Design and Implementation, 1997, `http://pages.cs.wisc.edu/~ramali/Papers/pldi97.ps`

[Rel] JDK 8, `http://openjdk.java.net/projects/jdk8/`

[Rob13] Ch. Robert, Default-Methoden in Java 8, in: JavaSPEKTRUM, 3/2013

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com

Oliver Tigges ist Principal Consultant bei der innoQ Deutschland GmbH. Er befasst sich seit zehn Jahren mit der Architektur und Realisierung von Webanwendungen und verteilten Unternehmensanwendungen. Sein besonderes Interesse gilt momentan den Themen DevOps, Continuous Delivery und Graphendatenbanken.
E-Mail: oliver.tigges@innoq.com