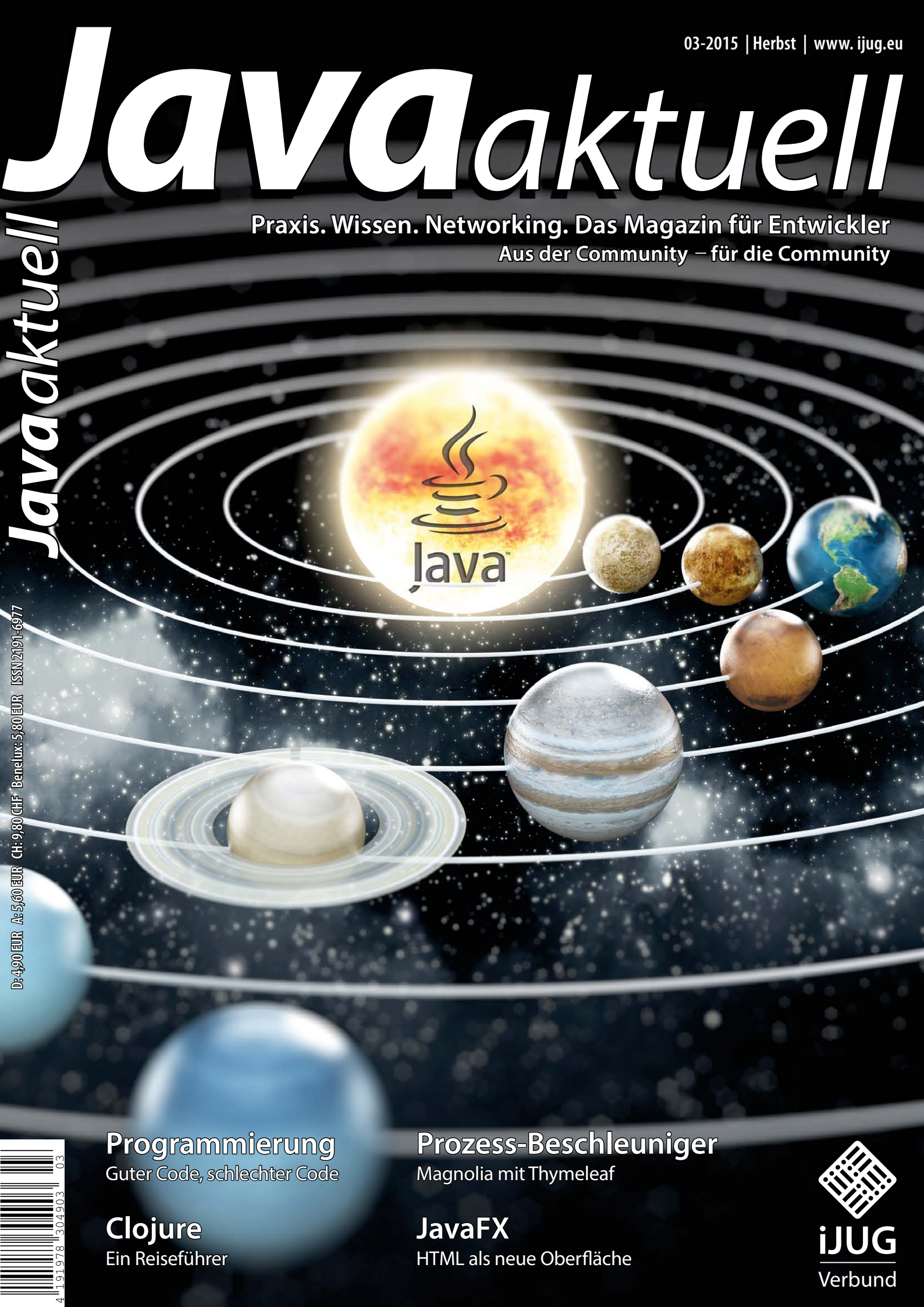
03-2015 | Herbst | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell



D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Programmierung
Guter Code, schlechter Code

Clojure
Ein Reiseführer

Prozess-Beschleuniger
Magnolia mit Thymeleaf

JavaFX
HTML als neue Oberfläche

iJUG Verbund

Kunstprojekt im JavaLand 2015

Seit Java 1.5 erlaubt die Java Virtual Machine die Registrierung sogenannter „Java-Agenten"

Bei Mgnolia arbeiten Web-Entwickler und CMS-Experten mit ein und demselben Quellcode

Ein Heim-Automatisierungs-Projekt

Aspektorientiertes Programmieren mit Java-Agenten

Rafael Winterhalter, Bouvet ASA

Aspektorientiertes Programmieren ist ein mächtiges Werkzeug, das jedoch nur wenige Programmierer nutzen. Seit Java 1.5 erlaubt die Java Virtual Machine die Registrierung sogenannter „Java-Agenten", die einen natürlichen Zugang zur aspektorientierten Programmierung mit Java bieten. Der Artikel wirft einen genaueren Blick auf solche Agenten und zeigt, wie sich Aspekte durch Bytecode-Manipulation einfach in ein Java-Programm integrieren lassen.

Obwohl den meisten Software-Entwicklern aspektorientierte Programmierung ein Begriff ist, spielt dieser Lösungsansatz im Java-Ökosystem eine eher untergeordnete Rolle – nicht zuletzt, weil viele Entwickler Sprach-Erweiterungen zur aspektorientierten Programmierung als zu umständlich empfinden. Denn selbst für triviale Anwendungsfälle erfordern diese Toolkits häufig Plug-ins für Entwicklungsumgebung und Build-Tool, da der Java-Compiler selbst keinen Aufsatzpunkt zur Implementierung von Aspekten bietet. Noch dazu ist die Sprache der „Pointcuts" und „Join Points", die in der aspektorientierten Programmierung üblich sind, vergleichsweise abstrakt und erfordert einige Einarbeitungszeit.

Im Kontrast zum Java-Compiler bietet die Java-Laufzeit-Umgebung, die Java Virtual Machine, allerdings einen natürlichen Ansatz zur Umsetzung von Aspekten in einer Java-Anwendung. Bereits seit Java 1.5 erlauben sogenannte „Java-Agenten" das dynamische Umschreiben von Java-Klassen und -Methoden sogar während der Laufzeit einer Java-Anwendung. Agenten werden dabei beim Start einer solchen Anwendung auf der Kommandolinie angegeben und sind ähnlich zu einer Bibliothek anschließend auf dem Klassenpfad verfügbar. Jeder Agent definiert dabei aber zusätzlich noch eine „premain"-Methode, die – wie der Methoden-Name bereits nahelegt – vor dem Start der „main"-Methode, der eigentlichen Java-Anwendung, aufgerufen wird. *Listing 1* zeigt, wie ein einfacher Agent damit implementiert werden könnte.

Würde die oben stehende Java-Klasse mit einer entsprechenden Manifest-Datei in eine „.jar"-Datei verpackt und mithilfe des Parameters „-javaagent" zu einer Anwendung hinzugefügt, dann wäre der Konsolenauswurf aus der „premain"-Methode also noch vor dem Start der eigentlichen Anwendung zu lesen.

Klassen durch Java-Agenten zur Laufzeit verändern

An sich ist der genannte Ansatz noch nicht sonderlich bemerkenswert, denn ein ähnliches Ergebnis ließe sich auch durch einfaches Aneinanderreihen von Methoden-Aufrufen erzielen. Ihren eigentlichen Wert entfalten Java-Agenten erst nach Definition eines zweiten, optionalen Methoden-Parameters vom Typ „Instrumentation". Instanzen dieses Interface erlauben es, Einfluss auf das Klassen-Laden in einer Java-Anwendung zu nehmen. So ist es beispielsweise möglich, den Klassenpfad auch noch nach Start einer Anwendung mit zusätzlichen „.jar"-Dateien zu erweitern. Sogar der Bootstrapklassen-Lader, der eigentlich nur Java-Basisklassen wie „String" oder „Object" lädt, kann auf diesem Weg mit weiteren Einträgen versehen werden.

Das wohl mächtigste Werkzeug, das durch das „Instrumentation"-Interface zugänglich gemacht wird, ist allerdings die Möglichkeit, einen „ClassFileTransformer" zu registrieren. Instanzen dieses Interface erlauben es, jegliche Klasse, die während der Ausführung eines Java-Programms geladen wird, beliebig zu verändern, noch bevor diese Klasse in das Programm geladen wird. Auch AspectJ und andere Sprach-Erweiterungen zur aspektorientierten Programmierung haben diese Möglichkeit für sich entdeckt, um sogenanntes „Load Time Weaving" als eine Alternative zum traditionellen „Build Time Weaving" umzusetzen. Zumindest das Umschreiben der Klassen kann damit zur Laufzeit erfolgen.

Um Nutzern eine Möglichkeit einzuräumen, eine bereits kompilierte Java-Klasse zur Laufzeit eines Java-Programms zu verändern, wird ein „ClassFileTransformer" jedes Mal aktiviert, wenn eine Klasse in einer Java-Anwendung durch einen „ClassLoader" geladen wird. Dies geschieht in der Regel bei der ersten Verwendung der Klasse, also wenn eine Zeile eines Java-Programms ausgeführt wird, in der diese Klasse zum ersten Mal genutzt wird. Daraus folgt auch, dass eine Klasse, die niemals benutzt wird, auch niemals geladen oder durch einen „ClassFileTransformer" verändert wird. Das hat den Vorteil, dass ein Java-Agent den Start eines Programms nicht erheblich verzögert, da unter anderem das Umschreiben einer Klasse nur bei Bedarf ausgeführt wird.

Die zu verändernde Java-Klasse wird einem „ClassFileTransformer" dabei in kompilierter Form als Byte-Array als ein Argument der einzigen Methode des Interface übergeben. Als Rückgabewert selbiger Methode erwartet der Agent dieselbe Klasse in der gewünschten angepassten Form, ebenfalls in Form eines Byte-Arrays, das die kompilier-

```
public class MeinAgent {
  public static void premain(String args) {
     System.out.println("Hallo! Ich bin ein Agent.");
  }
}
```

Listing 1

```
class NoOpClassFileTransformer implements ClassFileTransformer {
  @Override
  public byte[] transform(ClassLoader classLoader,
            String className,
            Class<?> classBeingRetransformed,
            ProtectionDomain protectionDomain,
            byte[] classFileBuffer) {
     return classFileBuffer;
  }
}
```

Listing 2

te Ausgabeklasse repräsentieren soll. Jede kompilierte Klasse wird dabei durch sogenannten „Java Bytecode" repräsentiert und nicht mehr durch den vielen Entwicklern gut bekannten Java-Quelltext.

Tatsächlich kann die Java Virtual Machine, trotz ihres Namens, nichts mit Java-Quelltextdateien anfangen. Die Programmiersprache Java ist durch den Java-Compiler vollständig vor der virtuellen Maschine verborgen, die einzig und allein Java-Bytecode ausführen kann. Nicht zuletzt deswegen ist es möglich, dass auch andere Programmiersprachen wie Scala oder Clojure nach Übersetzung in das Bytecode-Zwischenformat von der JVM ausgeführt werden können und auch mit in Java geschriebenen Bibliotheken interagieren. Im einfachsten Fall lässt sich ein „ClassFileTransformer" damit wie in *Listing 2* implementieren.

Wie der Name des „ClassFileTransformer" bereits andeutet, wird durch selbigen Transformator nichts bewirkt, da jede als „classFileBuffer" überreichte Klasse lediglich in ihrer bestehenden Form zurückgegeben wird. Alternativ dazu könnte der Transformator auch „null" zurückgeben, was der virtuellen Maschine zu verstehen gibt, dass eine zu ladende Klasse nicht verändert werden soll.

Zusätzlich zu der kompilierten Klasse erhält ein „ClassFileTransformer" auch den Namen der zu ladenden Klasse überreicht, sowie den „ClassLoader", der diese Klasse gerade zu laden versucht. Darüber hinaus wird der „ProtectionDomain" der Klasse mitgeteilt, welche etwaigen Sicherheitsbeschränkungen ein „SecurityManager" mit sich bringt, die auch ein „ClassFileTransformer" berücksichtigen soll. Für den Fall, dass eine bereits geladene Klasse durch die Java-HotSwap-Funktion neu definiert werden soll, würde noch dazu die zuvor geladene Klasse übergeben werden. Für erstmalig geladene Klassen ist dieser dritte Parameter allerdings immer „null".

```
MethodVisitor methodVisitor = ...
methodVisitor.visitIns(Opcodes.ICONST_1);
methodVisitor.visitIns(Opcodes.ICONST_2);
methodVisitor.visitIns(Opcodes.IADD);
```

Listing 3

```
class MeineKlasse {
  String hallo() { return null; }
}
```

Listing 4

```
TypePool typePool = TypePool.Default.ofClassPath();
new ByteBuddy()
  .redefine(typePool.describe("MeineKlasse"),
            ClassFileLocator.ForClassLoader.ofClassPath())
  .method(ElementMatchers.named("hallo"))
  .instrument(FixedValue.value("Hallo Welt!"))
  .make()
  .load(ClassLoader.getSystemClassLoader(),
        ClassLoadingStrategy.Default.INJECTION);
System.out.println(new MeineKlasse().hallo()); // Hallo Welt!
```

Listing 5

Java-Bytecode mit ASM direkt manipulieren

Als größtes Problem verbleibt allerdings der Umgang mit Java-Bytecode. Denn auch wenn dieser sich relativ nahe an der Programmiersprache Java orientiert, verbleibt dennoch eine Vielzahl an Unterschieden. Java-Bytecode ist als Annäherung an eine Maschinensprache konzipiert und führt seine Operationen ausschließlich auf einem (ebenso virtuellen) Stapel-Speicher aus. Dabei werden zu verarbeitende Werte auf dem genannten Stapel zunächst abgelegt und dann von anschließenden Operationen wieder ausgelesen.

Möchte man in Java beispielsweise die Zahlen „eins" und „zwei" addieren, werden beide Zahlen zunächst in Java-Bytecode auf den Stapelspeicher geschrieben. Nach diesem Ablegen der beiden Zahlen werden bei einer Addition beide Werte ausgelesen, um anschließend das Ergebnis zurück auf den Stapel zu schreiben. Auf gleiche Weise wird auch eine Methode aufgerufen, deren Argumente zunächst auf dem Stapel abgelegt werden müssen, bevor die Methode diese bei ihrer Ausführung auslesen kann.

Als Vorteil, eine virtuelle Maschine durch einen Stapelspeicher zu organisieren, ergibt sich eine geringere Länge des Bytecodes für diese virtuelle Maschinensprache. Zum Vergleich ergeben sich bei einer Maschinensprache, in der Werte in Registern abgelegt werden, häufig größere Programme, da Instruktionen sich nicht implizit auf den Wert an der Spitze des Stapels beziehen, sondern immer eine Registeradresse benötigen. Da Java als „Sprache des Internets" konzipiert ist, wurde es als Vorteil verstanden, dass bei geringem Platzverbrauch eine Java-Klasse schneller über ein Netzwerk übertragen würde, beispielsweise beim Laden eines Applets durch einen Browser. Doch gleichzeitig kann diese eher unübliche Abstraktion heute das direkte Arbeiten mit Java-Bytecode erschweren, auch wenn das natürlich möglich ist.

Eine beliebte Möglichkeit zur direkten Verarbeitung von Java-Bytecode ist die freie Bibliothek ASM, die unter anderem auch innerhalb des OpenJDK verwendet wird. Dabei wird jede Operation durch einen auch in der offiziellen JVM-Spezifikation üblichen Opcode zum Ausdruck gebracht. Dieser beschreibt dabei eine der virtuellen Maschine bekannte Stapel-Operation. Als Beispiel würden die folgenden Operationen eine Methode erzeugen, die die Zahlen „eins" und „zwei" addiert *(siehe Listing 3)*.

Wie zuvor beschrieben, müssen hierzu zunächst die Zahlen „eins" und „zwei" auf dem Stapelspeicher abgelegt werden, sodass die „IADD"-Operation sie anschließend auslesen kann, um dann die Summe der beiden Zahlen zurück auf den Stapel zu schreiben. Auf ähnliche Art und Weise kann auch eine kompilierte Methode aus einer „class"-Datei interpretiert werden. Dazu muss ein eigenes Objekt „MethodVisitor" übergeben werden, dem dann beim Lesen der Methode alle OpCodes in der vorgefundenen Reihenfolge als Argument übergeben werden. Auf diese Art könnte auch die durch einen „ClassFileTransformer" übergebene „class"-Datei interpretiert und umgeschrieben werden. Doch wie bereits zu erahnen, ist dieses Vorgehen häufig unnötigerweise umständlich und fehleranfällig. Direktes Manipulieren von Bytecode sollte deswegen für aspektorientierte Programmierung nur dann eingesetzt werden, wenn sehr spezielle Klassen-Veränderungen benötigt werden, die nicht durch ein weniger abstraktes API umgesetzt werden kann.

Bytecode-Manipulation mit Byte Buddy

Byte Buddy ist eine Bibliothek, die Bytecode-Manipulation auch ohne ein Verständnis des Java-Bytecode-Formats zugänglich macht. Byte Buddy basiert hierzu auf einer domänenspezifischen Sprache, mithilfe derer sich die Redefinition einer Klasse durch gewöhnlichen Java-Code ausdrücken lässt.

Als einfacher Anwendungsfall lässt sich zum Beispiel der Rückgabewert einer Methode zur Laufzeit auf einen festen Wert verändern. Für ein Beispiel soll dazu die einzige Methode der folgenden Klasse umgeschrieben werden *(siehe Listing 4)*.

Anstelle der „null"-Referenz soll die Methode mit Namen „hallo" aus „MeineKlasse" überschrieben werden, um einen festen Wert „Hallo Welt!" anstatt von „null" zurückzugeben. Mithilfe von Byte Buddy ist dies mit nur wenigen Linien Code möglich *(siehe Listing 5)*.

Da bisher kein Java-Agent zum Einsatz kommt, der vor dem Laden einer Klasse automatisch benachrichtigt wird, ist es dazu wichtig, dass „MeineKlasse" nicht bereits durch einen „ClassLoader" geladen wird, bevor die Klasse umgeschrieben wurde. Denn nach dem Laden einer Klasse erlaubt die Java Virtual Machine nur noch sehr eingeschränkte Änderungen an deren Funktionalität oder Struktur.

Um ein Laden der Klasse vor deren Umschreiben zu vermeiden, kann ein „TypePool" zum Einsatz kommen, der es erlaubt, eine Java-Klasse ähnlich wie durch das Java-Reflection-API zu beschreiben. Diese Beschreibung kann dann von Byte Buddy verarbeitet werden, um das Verhalten der Klasse zu verändern. Dabei werden zunächst eine oder mehrere Methoden durch einen „ElementMatcher" identifiziert. In oben stehendem Beispiel wird die „toString"-Methode anhand ihres Namens erkannt. Anschließend lässt sich das Verhalten dieser Methoden mithilfe einer Byte-Buddy-„Instrumentation" anpassen. Im Beispiel ist lediglich ein fester Wert durch einen „FixedValue" definiert, eine der vielen Implementierungen des Interface.

Die Rückgabe eines festen Werts wie „Hallo Welt!" ist dabei ein einfacher Anwendungsfall. In der aspektorientierten Programmierung soll das Verhalten einer Methode in der Regel aber dynamischer angepasst werden. Auch dies ist mithilfe von Byte Buddy einfach zu lösen, beispielsweise durch die Verwendung einer „MethodDelegation". Über eine solche Delegation erlaubt Byte Buddy, einen Methoden-Aufruf durch eine alternative Methode zu ersetzen. Dabei werden die Parameter-Werte der aufzurufenden Methode durch Annotationen bestimmt. Zum Beispiel ermöglicht die „@Origin"-Annotation, die umgeschriebene Methode abzufragen, wie der Interceptor im Beispiel vom *Listing 6* demonstriert.

Mithilfe des genannten Interceptor kann „MeineKlasse" nun entsprechend umgeschrieben werden, sodass anstelle von „methode" die „intercept"-Methode von „MeinInterceptor" aufgerufen wird. Letztere Methode wird dabei als Argument eine Referenz zu der ursprünglich aufgerufenen „methode" übergeben, wie das Beispiel in *Listing 7* verdeutlicht.

Ein einfacher Agent mit Byte Buddy

In Kombination mit der Möglichkeit, einen Java-Agenten zu registrieren, wird es plötzlich einfach, Aspekte auch ohne eine Sprach-Erweiterung wie AspectJ zu implementieren. Als Beispiel soll der Aufruf jeder Methode, die mit der folgenden „Loggen"-Annotation annotiert ist, auf der Konsole angezeigt werden *(siehe Listing 8)*.

Um dies mit Byte Buddy zu implementieren, muss zunächst ein entsprechender Interceptor für das Loggen definiert sein. Dieser empfängt als Argument eine Referenz der annotierten

Methode und schreibt deren Namen anschließend auf die Standard-Ausgabe *(siehe Listing 9)*.

Mithilfe des Interceptor kann, wie im vorherigen Abschnitt gezeigt, eine Methode umgeschrieben werden, sodass die Methode des „LogInterceptor" anstelle der eigentlichen Methode aufgerufen wird. Dies entspricht allerdings noch nicht der Anforderung, da der Interceptor zusätzlich zu und nicht anstatt der umgeschriebenen Methode aufgerufen werden soll. Auch dies ist mit Byte Buddy möglich, da viele der mitgelieferten Instrumentierungen miteinander verknüpfbar sind. Um nach Aufruf des Interceptor die ursprüngliche Methode aufzurufen, kann die Delegation mit einem solchen Aufruf durch „MethodDelegation.to(LogInterceptor.class).andThen(SuperMethodCall.INSTANCE)" verbunden werden. Die verknüpfte Instrumentierung ruft dabei lediglich die ursprüngliche Methode auf.

Um die Definition eines Agenten zu erleichtern, bietet Byte Buddy die Nutzung eines „AgentBuilder". Dieser erlaubt es zunächst, Klassen mithilfe eines „ElementMatcher" zu identifizieren, die vor dem Laden umgeschrieben werden sollen. Anschließend wird ein „Transformator" für diese Klassen registriert, der es ermöglicht, jede der zuvor identifizierten Klassen mithilfe der bereits bekannten domänenspezifischen Sprache umzuschreiben. Über einen solchen „AgentBuilder" kann ein Aspekt zum Loggen eines annotierten Methodenaufrufs nun einfach implementiert werden, wie das Beispiel in *Listing 10* zeigt.

Der oben stehende Aspekt zum Loggen eines Methodenaufrufs soll für sämtliche Klassen implementiert werden, aber nur für solche Methoden, die mit „Loggen" annotiert sind. Für diese Methoden soll dann zunächst der „LogInterceptor" und dann die ursprüngliche Methode aufgerufen werden.

Da Annotationen, die nicht auf dem Klassenpfad aufzufinden sind, von der Java-Laufzeit-Umgebung ignoriert werden und keine „ClassNotFoundException" verursachen, können der Agent, die Annotation und der „LogInterceptor" entfernt werden, ohne die Funktionalität der Anwendung zu stören. Genauso wenig muss die Anwendung vor dem Entfernen oder Hinzufügen des Agenten neu kompiliert werden. Die „Loggen"-Annotation kann aus diesem Grund sogar als „drop-in Feature" verwendet werden.

Natürlich bieten viele Rahmenwerke wie beispielsweise Spring bereits eine Möglichkeit, Aspekte zu implementieren. Außerdem ist es möglich, Aspekte mit Sprach-Erweiterungen wie zum Beispiel AspectJ zu implementieren. Durch Definition eines eigenen Agenten lassen sich Aspekte allerdings ohne viele Codezeilen und unabhängig von einer tiefer greifenden Infrastruktur umsetzen. Gerade Querschnittsanforderungen wie Logging, Sicherheit oder Caching sind dadurch auf einfache Art und Weise zu realisieren.

```
public class MeinInterceptor {
  public static String intercept(@Origin Method methode) {
     return "Hallo von " + methode.getName() + "!";
  }
}
```

Listing 6

```
TypePool typePool = TypePool.Default.ofClassPath();
new ByteBuddy()
  .redefine(typePool.describe("MeineKlasse"),
            ClassFileLocator.ForClassLoader.ofClassPath())
  .method(ElementMatchers.named("hallo"))
  .instrument(MethodDelegation.to(MeinInterceptor.class))
  .make()
  .load(ClassLoader.getSystemClassLoader(),
        ClassLoadingStrategy.Default.INJECTION);
System.out.println(new MeineKlasse().hallo()); // Hallo von methode!
```

Listing 7

```
@Retention(RetentionPolicy.RUNTIME)
@interface Loggen { }
```

Listing 8

```
public class LogInterceptor {
  public static void intercept(@Origin Method methode) {
     System.out.println(methode.getName() + " wurde aufgerufen!");
  }
}
```

Listing 9

```
public class LogAgent {
  public static void premain(String args,
                     Instrumentation instrumentation) {
     new AgentBuilder.Default()
        .rebase(ElementMatchers.any())
        .transform(
           (builder, typeDescription) -> return builder
              .method(ElementMatchers.isAnnotatedWith(Loggen.class))
              .intercept(MethodDelegation.to(LogInterceptor.class)
                    .andThen(SuperMethodCall.INSTANCE));
     ).installOn(instrumentation);
  }
}
```

Listing 10

Rafael Winterhalter
rafael.wth@gmail.com

Rafael Winterhalter lebt in Oslo und arbeitet dort als Softwareentwickler. Er beschäftigt sich intensiv mit der JVM und entwickelt aktiv das Open-Source-Projekt Byte Buddy (http://bytebuddy.net).

http://ja.ijug.eu/15/3/5