GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

3 / 2019

JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

#JAVAPRO

Horror Serialisierung

Dramatische Sicherheitslecks in Java

#JAVAPRO #CoreJava #Annotations

Deep-Dive into Annotations – Teil 3

Java-Annotationen sind ein mächtiges Sprachmerkmal, deren Interna allgemein nicht sonderlich bekannt sind. In Teil 3 unserer dreiteiligen Serie geht es um die Auswertung eigener Annotationen zur Laufzeit.

In Teil 1 unserer Serie wurden diverse Verwendungszwecke für Annotationen aufgezeigt und die fünf Java-SE-Standardannotationen detailliert diskutiert. Teil 2 führte aus, wie eigene Annotationstypen programmiert werden können. Dabei wurden insbesondere die zahlreichen Optionen bei der Konfiguration sowie diverse Besonderheiten betrachtet. Die Syntax der Definition eigener Annotationen ist wahrlich sehr speziell und passt so gar nicht ins Bild der ansonsten eigentlich schönen und konsistenten Java-Syntax.

Im hier vorliegenden dritten und letzten Teil kommen wir zur Krönung des Einsatzes von Annotationen. Die Deklaration und das Anbringen von Annotationen macht nur Sinn, wenn sich

Autor:

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Software-Entwicklung, -Schulung und -Beratung v. a. für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.

christian.heitzmann@simplexacode.ch
https://www.simplexacode.ch

diese auch wieder auslesen lassen. Erinnern wir uns, dass es sich gemäss offizieller Definition bei Annotationen lediglich um Marker handelt, die Informationen mit einem Programmkonstrukt verknüpfen, aber keinen Effekt zur Laufzeit haben. Annotationen können also von sich selbst aus nichts „machen". Daraus ergeben sich drei Schlussfolgerungen:

1. Zum Auslesen von Annotationen liegt der Einsatz von Reflection auf der Hand.
2. Es lassen sich einzig die Existenz bzw. Nichtexistenz von Annotationen feststellen und die in den Annotationen enthaltenen Parameter auslesen. Mit anderen Worten, Annotationen lassen sich nicht ausführen.
3. Soll das Vorhandensein einer Annotation mit irgendwelchen Aktionen verknüpft werden, so hat dies von aussen zu geschehen. Sämtliche Programmlogik muss in normalem Java-Code in diesen „Erkennungsroutinen" hinterlegt sein.

Die Optionen zur Definition eigener Annotationen sind zahlreich. Dementsprechend zahlreich sind auch die verschiedenen Arten, wie sich Annotationen in eigenen Programmen auslesen lassen. Dazu werden wir nachfolgend alle typischen Fälle betrachten, wie sich welche Art von Annotationen auslesen lässt.

Deklaration verschiedener Annotationstypen

Um die unterschiedlichen Arten zum Auslesen von Annotationen aufzeigen zu können, benötigen wir zuerst verschiedene eigene Annotationstypen. Folgende Annotationen stellen eine Zusammenfassung der in Teil 2 erläuterten Optionen dar:

(Listing 1)

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface NoValueAnnotation {}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface SingleValueAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MultiValueAnnotation {
  public int[] value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(RepeatableValueAnnotationContainer.class)
@interface RepeatableValueAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RepeatableValueAnnotationContainer {
  public RepeatableValueAnnotation[] value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface SingleNamedValueAnnotation {
  public String name();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MultiNamedValueAnnotation {
  public String first();
  public String second();
  public String third();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface EnumValueAnnotation {
  public static enum GreekLetter {
    ALPHA, BETA, GAMMA
  }
  public GreekLetter value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface NonInheritedAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@interface InheritedAnnotation {
  public int value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
@interface ParameterAnnotation {}
```

- Um zur Laufzeit Zugriff auf die Annotationen zu haben, müssen diese allesamt mit **`@Retention(RetentionPolicy.RUNTIME)`** versehen werden. **`RetentionPolicy.SOURCE`** wird wohl nur für eigene Compiler oder Tools, interessant sein, die direkt auf Source-Code-Ebene arbeiten, zum Beispiel so etwas wie Javadoc. **`RetentionPolicy.CLASS`** funktioniert entsprechend nur für eigene Tools, die direkt mit dem Bytecode der Klassen arbeiten, zum Beispiel Code-Obfuscatoren oder allenfalls eigene Klassenlader. Das Gros der Anwendungsfälle wird das Auslesen der Annotationen zur Laufzeit ausmachen, und dafür kommt nur **`RetentionPolicy.RUNTIME`** in Frage.

- Aus Gründen der Einfachheit wurden die meisten Annotationstypen mit **`@Target(ElementType.METHOD)`** versehen, lassen sich also nur zur Annotation von Methoden

verwenden. Dies ist keinesfalls einschränkend zu verstehen. Das Annotieren zum Beispiel von Klassen (**`ElementType.TYPE`**), Attributen (**`ElementType.FIELD`**) oder Methoden-Parametern (**`ElementType.PARAMETER`**) wäre in den meisten Fällen analog möglich, sofern die passenden @Target konfiguriert und die entsprechenden Reflection-Aufrufe vorgenommen werden. Um den Rahmen nicht zu sprengen, wurde darauf in diesem Artikel verzichtet.

- **`NoValueAnnotation`** ist eine leere Marker-Annotation. **`SingleValueAnnotation`** ist eine solche mit einem einzigen Wert, in diesem Fall einem Integer mit dem Standardnamen **`value`**. Wir erinnern uns: Der Name **`value`** muss bei der Anwendung einer Annotation nicht explizit genannt werden, alle anderen Namen hingegen schon.

- **`MultiValueAnnotation`** ist eine Annotation mit mehreren Werten gleichen Typs und dem Standardnamen value, das heisst, auch hier können bei der Anwendung die Werte einfach in geschweiften Klammern mitgegeben werden, ohne dass **`value`** explizit dazugeschrieben werden muss. **`RepeatableValueAnnotation`** ist ebenfalls in der Lage, mehrere Werte gleichen Typs aufzunehmen. Im Gegensatz zu **`MultiValueAnnotation`** muss dies aber mit mehreren Einzelaufrufen geschehen. Um dies zu erreichen, wurde die Annotation wiederum mit **`@Repeatable`** annotiert, einer neuen Meta-Annotation seit Java 8. **`@Repeatable`** funktioniert nur in Verbindung mit einem Container, der hier **`@RepeatableValueAnnotationContainer`** genannt wurde und nicht zum direkten Einsatz als Annotation gedacht ist, sondern seinen Dienst nur hinter den Kulissen verrichten soll.

- **`SingleNamedValueAnnotation`** und **`MultiNamedValueAnnotation`** sind Annotationen mit einem respektive mehreren Werten, die jedoch nicht standardmäßig value, sondern individuell benannt sind (**`name`** respektive **`first`**, **`second`** und **`third`**).

- **`EnumValueAnnotation`** zeigt die Deklaration und später den Einsatz einer Annotation, die ihren eigenen Aufzählungstyp enthält (in diesem Beispiel **`GreekLetter`**).

- **`NonInheritedAnnotation`** und **`InheritedAnnotation`** unterscheiden sich einzig durch die **`@Inherited`** Meta-Annotation. Zur Erinnerung: **`@Inherited`** funktioniert nur bei Klassen (daher hier auch das **`@Target(ElementType.TYPE)`**), nicht bei Interfaces und erst recht nicht bei Methoden, Attributen oder dergleichen.

- **`ParameterAnnotation`** annotiert, wie der Name bereits verrät, Methodenparameter, um abschliessend aufzeigen zu können, wie zum Beispiel die Argumente einer Methode abgefragt werden können.

Anbringung der Annotationen

Das Anbringen der im vorherigen Abschnitt beschriebenen Annotationen erklärt sich nun von selbst:

(Listing 2)

```
@NonInheritedAnnotation(42)
@InheritedAnnotation(21)
class Super {

  public void methodWithNoAnnotation() {}

  @NoValueAnnotation
  public void methodWithNoValueAnnotation() {}

  @SingleValueAnnotation(42)
  public void methodWithSingleValueAnnotation() {}

  @MultiValueAnnotation( { 1, 23, 456, 7890 } )
  public void methodWithMultiValueAnnotation() {}

  @RepeatableValueAnnotation(1)
  @RepeatableValueAnnotation(23)
  @RepeatableValueAnnotation(456)
  @RepeatableValueAnnotation(7890)
  public void methodWithRepeatableValueAnnotation() {}

  @SingleNamedValueAnnotation(name = "forty-two")
  public void methodWithNamedSingleValueAnnotation() {}

  @MultiNamedValueAnnotation(first = "one",
                             second = "two",
                             third = "three")
  public void methodWithMultiNamedValueAnnotation() {}

  @EnumValueAnnotation(EnumValueAnnotation.GreekLetter.GAMMA)
  public void methodWithEnumValueAnnotation() {}

  public void methodWithAnnotatedParameter
      (Object parameterWithNoAnnotation,
       @ParameterAnnotation Object parameterWithAnnotation) {}
}

final class Sub extends Super {}
```

Praktisch alle Annotationen werden auf oder in der Klasse **`Super`** angewendet. Ganz unten im Quellcode findet sich noch eine Klasse **`Sub`**, die von **`Super`** erbt und später das Verhalten der **`@Inherited`** Klassenannotation demonstrieren soll. Im folgenden Abschnitt geht es darum, die verschiedenen Arten in eigenen Programmen abzufragen.

Annotation Processor

Wir möchten nun einen sogenannten Annotationsprozessor schreiben, also ein Programm, welches unsere eigenen Annotationen ausliest und ausgibt, die wir an den analog benannten Methoden, an der Klasse und in einem Fall an den Methodenparametern angebracht haben. Enthalten die Annotationen Werte, so werden diese ebenfalls auf der Konsole ausgegeben. Die Hauptklasse mit ihrer **`main`**-Methode sieht wie folgt aus:

(Listing 3)

```
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public final class AnnotationProcessor {

  public static void main(String[] args) {
    Class<Super> superClass = Super.class;
    Class<Sub> subClass = Sub.class;
    AnnotationProcessor ap = new AnnotationProcessor();
    ap.printMethodsWithNoAnnotation(superClass);
    ap.printMethodsWithNoValueAnnotation(superClass);
    ap.printMethodsWithSingleValueAnnotation(superClass);
    /* [...] Other method calls omitted. */
```

Methoden ohne Annotation

(Listing 4)

```
  void printMethodsWithNoAnnotation(Class<?> clazz) {
    System.out.println("Methods With No Annotation");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      Annotation[] annotations
          = currentMethod.getAnnotations();
      if (annotations.length == 0) {
        String methodName = currentMethod.getName();
        System.out.format("  %s%n", methodName);
      }
    }
  }
```

Zur Vereinfachung haben wir bislang meistens nur Methoden annotiert. Im ersten Beispiel geht es darum, alle Methoden auszugeben, die über keine Annotation verfügen. Start einer solchen Suche ist immer das Klassenobjekt, welches man durch **`.class`** oder **`getClass`** erhält. Hat man ein solches Klassenobjekt, in (Listing 4) als **`clazz`** bezeichnet, erhält man via **`getDeclaredMethods`** ein Array von **`Methods`**. Im Gegensatz zu **`getMethods`** gibt **`getDeclaredMethods`** zum einen auch die privaten Methoden einer Klasse zurück, zum anderen sind alle Methoden der Oberklasse(n) (inklusive **`Object`**) ausdrücklich nicht Bestandteil dieser Auflistung. Analog erhält man mit **`Method#getAnnotations`** ein Array aller Annotationen einer Methode. Ist dieses Array leer (also **`length == 0`**), dann folgt daraus, dass diese Methode nicht annotiert wurde. Sie ist damit genau das, was im hier vorliegenden Fall gesucht wird und ihr Methodenname (**`Method#getName`**) wird auf der Konsole ausgegeben.

Methoden mit Annotation ohne Wert

(Listing 5)

```
  void printMethodsWithNoValueAnnotation(Class<?> clazz) {
    System.out.println("Methods With @NoValueAnnotation");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      NoValueAnnotation annotation
          = currentMethod.getAnnotation
          (NoValueAnnotation.class);
      if (annotation != null) {
        String methodName = currentMethod.getName();
        System.out.format("  %s%n", methodName);
      }
    }
  }
```

Wird nach einer ganz bestimmten Annotation gesucht, so bietet sich **`Method#getAnnotation(Class<T extends Annotation>)`** an. Als Parameter wird der gesuchte Annotationstyp mitgegeben. Wurde die entsprechende Annotation angebracht, so gibt der Aufruf ebendiese Annotation zurück; wenn nicht, dann ist der Rückgabewert **`null`**. Eine anschliessende Überprüfung auf **`null`** ist also immer obligatorisch, wenn man keine Laufzeitfehler riskieren will. In (Listing 5) wird gezielt nach der **`@NoValueAnnotation`** gesucht. Wird sie gefunden, dann wird der Name der Methode, an der sie angebracht wurde, auf der Konsole ausgegeben.

Methoden mit Annotation mit nur einem Wert

(Listing 6)

```
void printMethodsWithSingleValueAnnotation(Class<?> clazz) {
  System.out.println
      ("Methods With @SingleValueAnnotation(int)");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    SingleValueAnnotation annotation
        = currentMethod.getAnnotation
        (SingleValueAnnotation.class);
    if (annotation != null) {
      int value = annotation.value();
      System.out.format("  %s | value=%d%n",
                        methodName, Integer.valueOf(value));
    }
  }
}
```

Das Vorgehen zum Detektieren der gesuchten Annotation (hier **`@SingleValueAnnotation`**) ist aus dem vorherigen Code-Beispiel bekannt und wird sich von nun an auch nicht mehr ändern. Neu ist in (Listing 6), wie sich der Wert **`value`** der Annotation abfragen lässt. Da bereits der konkrete Annotationstyp vorliegt, geschieht dies ganz einfach mit dem Aufruf von **`.value()`**. Im Beispiel wird der so ausgelesene Integer-Wert zusammen mit dem Methodennamen auf der Konsole ausgegeben.

Methoden mit Annotation mit mehreren Werten

(Listing 7)

```
void printMethodsWithMultiValueAnnotation(Class<?> clazz) {
  System.out.println
      ("Methods With @MultiValueAnnotation(int...)");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    MultiValueAnnotation annotation
```

```
        = currentMethod.getAnnotation
        (MultiValueAnnotation.class);
    if (annotation != null) {
      int[] values = annotation.value();
      System.out.format("  %s | values=%s%n",
                        methodName, Arrays.toString(values));
    }
  }
}
```

Bei Annotationen, die mehrere Werte gleichen Typs aufnehmen können, liefert die Methode **`.value()`** ein Array statt eines einzelnen Wertes zurück. Der anschließende Zugriff auf die Werte des Arrays ist trivial.

Methoden mit wiederholter Annotation mit Wert

(Listing 8)

```
void printMethodsWithRepeatableValueAnnotation
    (Class<?> clazz) {
  System.out.println
      ("Methods With @RepeatableValueAnnotation(int)*");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    RepeatableValueAnnotation[] annotations
        = currentMethod.getAnnotationsByType
        (RepeatableValueAnnotation.class);
    for (RepeatableValueAnnotation currentAnnotation
         : annotations) {
      int value = currentAnnotation.value();
      System.out.format("  %s | value=%d%n",
                        methodName, Integer.valueOf(value));
    }
  }
}
```

Zwischen Annotationen mit mehreren Werten und **`@Repeatable`** Annotationen mit jeweils einem Wert, gibt es nicht nur beim Anbringen, sondern auch beim Auslesen leichte Unterschiede. Da nun mehrere Annotationen eines gesuchten gleichen Typs vorkommen können, funktioniert **`Method#getAnnotation`** nicht. Stattdessen muss **`Method#getAnnotationsByType(Class<T extends Annotation>)`** aufgerufen werden, welches nun ein Array von Annotationen des gesuchten Typs zurückgibt, sofern vorhanden. Im schlimmsten Fall ist dieses Array einfach leer; eine Überprüfung auf **`null`** kann entfallen. Um die Werte auszulesen, kann einfach dieses Array iteriert werden. Die darin enthaltenen Annotationen verhalten sich gleich wie die Annotationen mit nur einem Wert.

Methoden mit Annotation mit nur einem benannten Wert

(Listing 9)

```
void printMethodsWithSingleNamedValueAnnotation
    (Class<?> clazz) {
  System.out.println
      ("Methods With @SingleNamedValueAnnotation(String)");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    SingleNamedValueAnnotation annotation
        = currentMethod.getAnnotation
        (SingleNamedValueAnnotation.class);
    if (annotation != null) {
      String name = annotation.name();
      System.out.format("  %s | name=%s%n", methodName, name);
    }
  }
}
```

Wir haben zuvor bereits auf den Parameter **`value`** zugegriffen. Sind die Werte einer Annotation anders benannt, so erfolgt ihr Zugriff analog anhand dieses Namens. Vorangehend wurde der Wert einer Annotation **`name`** genannt. Der Zugriff darauf erfolgt also einfach über **`.name()`**.

Spätestens an dieser Stelle wird nun klar, wieso Annotationen ähnlich wie Interfaces implementiert werden und wieso die „Attribute", sprich die Werte oder Parameter einer solchen Annotation immer als Anhang von Zugriffsmethoden, also mit einem nachfolgenden Klammerpaar deklariert werden müssen. Zur Laufzeit stellt diese Annotation nach außen hin nichts anderes als ein Interface mit ebendiesen Zugriffsmethoden dar.

Methoden mit Annotation mit mehreren benannten Werten

(Listing 10)

```
void printMethodsWithMultiNamedValueAnnotation
      (Class<?> clazz) {
    System.out.println
        ("Methods With @MultiNamedValueAnnotation(String)");
    for (Method currentMethod : clazz.getDeclaredMethods()) {
      String methodName = currentMethod.getName();
      MultiNamedValueAnnotation annotation
          = currentMethod.getAnnotation
          (MultiNamedValueAnnotation.class);
      if (annotation != null) {
        String first = annotation.first();
        String second = annotation.second();
        String third = annotation.third();
        System.out.format("  %s |"
                          + " first=%s, second=%s, third=%s%n",
                          methodName, first, second, third);
      }
    }
  }
```

Egal ob es sich um eine Annotation mit einem oder mehreren benannten Werten handelt, der Zugriff darauf erfolgt einheitlich und selbstklärend über den Namen des Wertes als Methodenaufruf, in **(Listing 10)** mit **`.first()`**, **`.second()`** und **`.third()`**.

Methoden mit Annotation mit Wert als Aufzählungstyp

(Listing 11)

```
void printMethodsWithEnumValueAnnotation(Class<?> clazz) {
  System.out.println
      ("Methods With @EnumValueAnnotation(GreekLetter)");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    EnumValueAnnotation annotation
        = currentMethod.getAnnotation
        (EnumValueAnnotation.class);
    if (annotation != null) {
      EnumValueAnnotation.GreekLetter value
          = annotation.value();
      System.out.format("%s | name=%s%n", methodName, value);
    }
  }
}
```

Auch der Zugriff auf Werte einer Annotation, die einen Aufzählungstyp darstellen, ist nicht besonders schwer. Wenn der Aufzählungstyp wie im vorliegenden Code-Beispiel direkt in der Annotation definiert ist, so erhält man auch nur über diese Annotation Zugriff auf diesen Typ. In (Listing 11) ist dies gut ersichtlich an **`EnumValueAnnotation.GreekLetter`**.

Klassen mit @Inherited-Annotation

(Listing 12)

```
void printClassesWithNonInheritedAnnotation
    (Class<?>... classes) {
  System.out.println("Classes With @NonInheritedAnnotation");
  for (Class<?> currentClass : classes) {
    String className = currentClass.getSimpleName();
    NonInheritedAnnotation annotation
        = currentClass.getAnnotation
        (NonInheritedAnnotation.class);
    if (annotation != null) {
      int value = annotation.value();
      System.out.format("  %s | value=%d%n",
                        className, Integer.valueOf(value));
    }
  }
}

void printClassesWithInheritedAnnotation
    (Class<?>... classes) {
  System.out.println("Classes With @InheritedAnnotation");
  for (Class<?> currentClass : classes) {
    String className = currentClass.getSimpleName();
    InheritedAnnotation annotation
        = currentClass.getAnnotation
        (InheritedAnnotation.class);
    if (annotation != null) {
      int value = annotation.value();
      System.out.format("  %s | value=%d%n",
                        className, Integer.valueOf(value));
    }
  }
}
```

Wechseln wir von annotierten Methoden zu annotierten Klassen. Die beiden Codebeispiele in (Listing 12), die sich nur am Annotationstyp unterscheiden, sollen demonstrieren, welche Auswirkung die **`@Inherited`** Meta-Annotation auf die Vererbung von Annotationen hat. Die Ausgabemethoden werden jeweils sowohl mit der Klasse **`Super`** als auch mit der Unterklasse **`Sub`** aufgerufen. Wie zu erwarten, ist die InheritedAnnotation sowohl in **`Super`** als auch in **`Sub`** zugreifbar, die **`NonInheritedAnnotation`** hingegen nur in **`Super`**.

Methodenparameter mit Annotation

(Listing 13)

```
void printMethodParametersWithAnnotation(Class<?> clazz) {
  System.out.println
      ("Method Parameters With @ParameterAnnotation");
  for (Method currentMethod : clazz.getDeclaredMethods()) {
    String methodName = currentMethod.getName();
    Parameter[] parameters = currentMethod.getParameters();
    for (Parameter currentParameter : parameters) {
      ParameterAnnotation annotation
          = currentParameter.getAnnotation
          (ParameterAnnotation.class);
      if (annotation != null) {
        String parameterName = currentParameter.getName();
        System.out.format("  %s.%s%n",
                          methodName, parameterName);
      }
    }
  }
}
```

Schauen wir uns zum Abschluss noch an, wie annotierte Methodenparameter ausgelesen werden können. Mit **`Method#getParameters`** lässt man sich via Reflection ein **`Parameter`** Array geben, welches anschließend iteriert wird. Selbsterklärend erhält man via **`Parameter#getAnnotation(Class<T extends Annotation>)`** Zugriff auf eine gesuchte Parameter-Annotation, sofern vorhanden. Das Auslesen ihrer Werte erfolgt auf die bekannte Art.

Fazit:

Die Literatur zum Thema Annotationen ist recht begrenzt und die Syntax sehr gewöhnungsbedürftig. Dennoch zeigt sich auf beeindruckende Weise, wie mächtig und flexibel Annotationen in Java eingesetzt werden können. Unserer 3-teiligen Serie soll dazu beitragen, anfängliche Berührungsängste gegenüber Annotationen zu beseitigen und ihren sinnvollen Einsatz in Softwareprojekten fördern.