www.javaspektrum.de

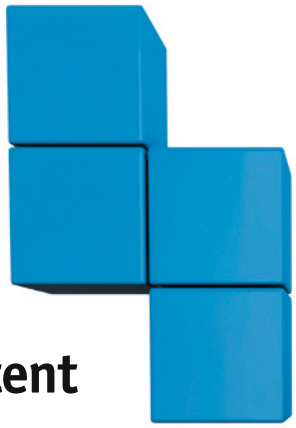
Ausgabe 6/2022
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 06

JavaSPEKTRUM

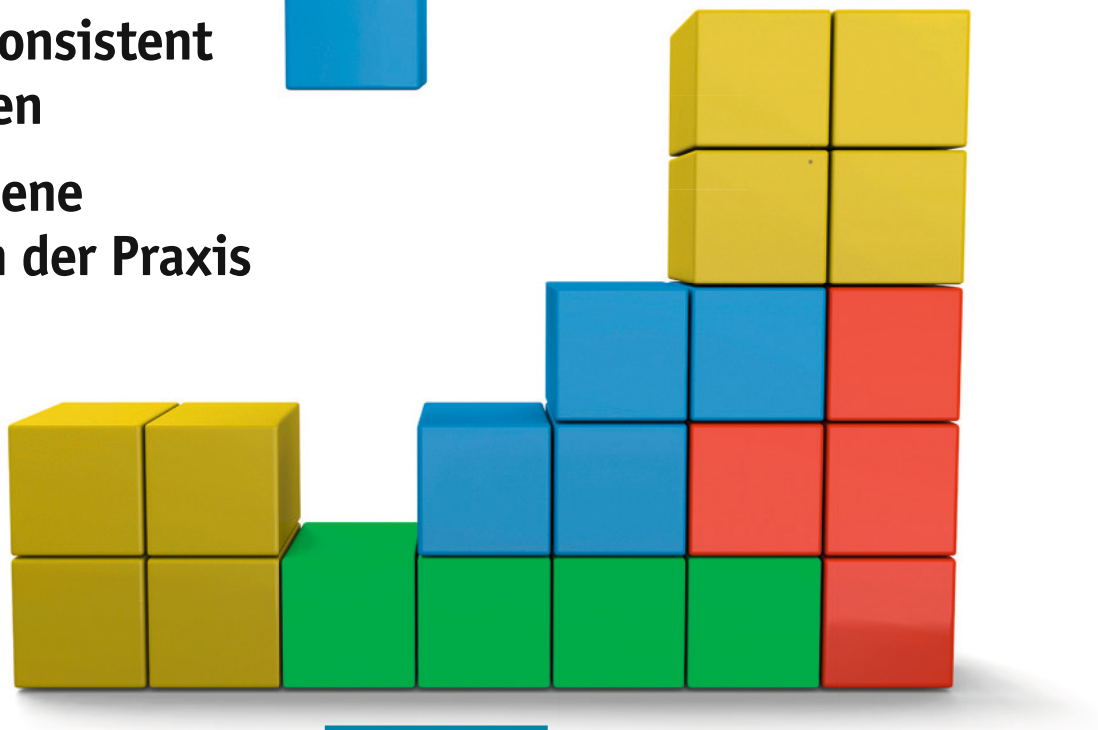
Magazin für professionelle Entwicklung und digitale Transformation

Java-Architekturen – komplexe Systeme entwerfen



Mit Javadoc konsistent kommunizieren

Event-getriebene Architektur in der Praxis



Interview

Interview mit Dr. Martin Nusswald, CIO von thyssenkrupp Materials Services über Digitalisierung und was unter „Materials as a Service“ zu verstehen ist

Fachthemen

Ohne Rast zu Rust – Rust für Java-Entwickler

Nachhaltigkeit im Software-Engineering – Teil 4: Zombies

zum Inhalt

Java-Architekturen – komplexe Systeme entwerfen

G30325

Konsistent und quellcodenah kommunizieren

Javadoc mit Stil

Christian Heitzmann

Javadoc hat sich als quellcodenahe API-Dokumentationslösung in den letzten zwei Jahrzehnten sehr bewährt. Der Einsatz seiner Dokumentationskommentare bedarf aber auch etwas Detailkenntnis. Wer um die entscheidenden Javadoc-Feinheiten Bescheid weiß, fühlt sich sicherer, kann konsistentere Dokumentationen erstellen und damit die Softwarequalität in mehrerer Hinsicht verbessern.

Javadoc gibt es seit den Anfangstagen von Java und ist seitdem ein treuer Begleiter nicht nur des JDKs selbst, sondern auch diverser Drittbibliotheken. Ein Mindestmaß an Disziplin vorausgesetzt, macht die unmittelbare Nähe der Dokumentationskommentare zum Quellcode ein Auseinanderdriften dieser beiden deutlich unwahrscheinlicher. Zumindest ist es mit dieser Quellcodenähe deutlich schwieriger, den Zustand des gefürchteten „Wiki Rot" zu erreichen: veraltete, überformelle, nutzlose, lästige oder gar nicht vorhandene Seiten betreffend Softwarearchitektur auf dem Firmen-Wiki oder gar als Word-Dateien auf SharePoint.

Selbstverständlich ist Dokumentation ein großes Fachgebiet. Es gibt verschiedene Arten von Dokumentation, die sich insbesondere in ihren Zielgruppen und Sichtweisen unterscheiden. Auf diese kann im Rahmen dieses Artikels leider nicht eingegangen werden, aber wer sich darüber informieren will (was ich sehr empfehle!), findet mit einer Suche nach *Gernot Starke* oder *Stefan Zörner* sicherlich die im deutschen Sprachraum wohl bekanntesten Personen und ihre dazugehörigen Bücher und Kurse. Wer einen guten ersten Überblick will, dem sei [Zör22] empfohlen.

Es wäre natürlich anmaßend, zu behaupten, dass Javadoc die Eier legende Wollmilchsau der Dokumentationslösungen wäre. Im Rahmen dieses Artikels werde ich nur Softwareentwickler als Verfasser und als Leser von technischer Dokumentation betrachten; also keine Endkunden, keine POs, keine Vertriebler und keine Manager. Meine Erfahrung (und hoffentlich auch die der softwareent-

Christian Heitzmann ist Java-, Python- und Spring-zertifizierter Softwareentwickler mit einem CAS in Machine Learning und Inhaber der SimplexaCode AG in Luzern. Er entwickelt seit über 20 Jahren Software und gibt seit über 12 Jahren Unterricht und Kurse im Bereich der Java- und Python-Programmierung, Mathematik und Algorithmik.
E-Mail: christian.heitzmann@simplexacode.ch

wickelnden Leser) zeigt aber, dass sich mit Javadoc in der Entwickler-zu-Entwickler-Kommunikation doch ein entscheidend großer Teil der Bedürfnisse abdecken lässt. Wer also heute noch dokumentationstechnisch auf einer grünen Wiese steht: Javadoc wäre der ideale Anfang.

Gründe für (quellcodenahe) Dokumentation

Die vielen Gründe für umfassende Dokumentation eigens entwickelter Software liegen theoretisch auf der Hand und werden unisono in der populären Entwicklungsliteratur propagiert. Dennoch scheint keine Arbeit so unbeliebt zu sein und als „unnötig" abgestempelt zu werden wie die des Dokumentierens.

Machen wir uns nichts vor: Dokumentationsschulden – das heißt keine, wenig oder veraltete Dokumentation – sind **technische Schulden**. Eine gute Dokumentation bildet die Grundlage für weitere Analysen und Verbesserungen der Softwarearchitektur [Lil20]. Ohne Dokumentation fällt diese entscheidende Grundlage weg.

Software wird in der Regel viel öfter gelesen als geschrieben [Ull22]. Ich persönlich nenne das **„Multiplikator"**: Wenn ich beim Entwickeln 30 zusätzliche Minuten in eine gute Klassen- oder Methodenbeschreibung investiere, dafür aber im Gegenzug auf die nächsten 10 Jahre gesehen 20 weiteren Mitarbeitern mühsame und fehleranfällige Reverse-Engineering-Arbeit (mit jeweils über 30 Minuten Aufwand) ersparen kann, dann geht die Rechnung klar auf. Ganz zu schweigen von vielen (teuren) Programmierfehlern, die so bestenfalls vermieden werden können, weil in der Dokumentation bereits auf notwendige Vorbedingungen, Seiteneffekte oder andere Stolpersteine ausdrücklich hingewiesen wird.

Bis jetzt haben wir bei den von uns verwendeten Bibliotheken doch auch immer von deren ausführlichen (Javadoc-)APIs profitiert: das JDK selbst, JUnit, das Spring-Framework, Google Guava, Apache Commons, oder in anderen Sprachen Angular, NumPy, Tensorflow und viele mehr. Bei all diesen **populären Bibliotheken** ist es doch völlig selbstverständlich, sich einzig an deren API-Dokumentation zu orientieren. Keiner jener Bibliotheksentwickler wäre jemals auf die Idee gekommen zu sagen: *„The source code is the best documentation, that's why we don't publish any API documentation."* Wenn sie es so gelebt hätten, wäre die Bibliothek nicht populär geworden. So einfach ist das.

Wieso sollte also für eigene Software ein anderer Grundsatz gelten? Der Mensch ist schließlich kein Compiler. Wer von seinen Mitarbeitern erwartet, dass sie den eigenen Code in einer Sisyphusarbeit **gedanklich „dekompilieren"**, nur weil er oder sie selbst zu faul oder nicht in der Lage ist, ihn angemessen zu dokumentieren, verhält sich per Definition asozial. Ganz zu schweigen von den horrenden Kosten, die dabei für den Arbeitgeber oder den Kunden entstehen, weil die Reverse-Engineering-Arbeit ja mit jedem weiteren Mitarbeiter und nach Eintreten des Vergessens (in der Regel ein paar Tage bis wenige Wochen) wiederholt werden muss.

Den Code als primäre „Dokumentations"-Quelle zu betrachten, widerspricht auch den grundlegenden Prinzipien der objektorientierten Programmierung: **Datenkapselung und Geheimnisprinzip**. Eine der großen Errungenschaften der Softwareentwicklung der letzten Jahrzehnte ist die Aufteilung in öffentliche (`public`) und private (`private`) Abschnitte und damit verbundene klare Schnittstellen (`interfaces`) und somit „Verträge" gegenüber den Anwendern. Hinzu kommen diverse Sicherheitsnetze, die (versehentliches) Überschreiben von Daten oder unzulässige Aufrufe von Methoden unterbinden. Wieso sollten diese Prinzipien, die zur (technischen) Entwicklungs- und Laufzeit gelten, nicht auch zur (menschlichen) Lese- und Analysezeit gelten?

An dieser Stelle kommt meist der erste Einwand, dass mit Clean Code oder Test-Driven Development (TDD) oder Extreme Programming (XP) – you name it –„ wie sie es bei sich im Haus praktizieren, der Quellcode doch selbstsprechend und klar genug sei, und es daher keinen Grund gebe, darüber hinaus noch etwas zu kommentieren oder zu dokumentieren. Das konnte ich aus meinen Beobachtungen bis heute noch *nie* unterstreichen.

Es gibt tatsächlich diverse Aspekte, die sich *nicht* alleine mit Quellcode ausdrücken lassen: Vorbedingungen, Invarianten, Nachbedingungen, Seiteneffekte, Threadsicherheit, Möglichkeiten und Verhalten bei Vererbung, Idempotenz, Zustandsbehaftung oder -losigkeit, Erlaubnis und Verhalten bei Transaktionen, Begründungen für Designentscheide (warum so und nicht anders?), Einbettung und Verhalten im großen Ganzen, Anwendungsbeispiele beziehungsweise vorgesehene Anwendungsfälle, Referenzen auf andere Module, bekannte Bugs, offene Baustellen (TODOs) und vieles mehr. Wenn sich Klassen oder Methoden nicht einfach dokumentieren lassen, dann sind sie entweder zu groß, zu kompliziert oder die Entwickler haben sie selbst nicht verstanden. Alle drei Gründe sprechen dafür, noch einmal gründlich über die Bücher zu gehen.

Auch das **Agile Manifest** (*„Working software over comprehensive documentation"*) ist leider keine Ausrede, auf Dokumentation zu verzichten – das wäre auch zu schön gewesen, oder? Denn der Satz geht noch weiter: *„[...] while there is value in the items on the right, we value the items on the left more."* [Manifesto] Es war und ist also nirgendwo die Rede davon, dass es bei agilen Arbeitsweisen wie Scrum keine Dokumentation mehr braucht! Natürlich würde auch ich im Fall der Fälle einer funktionierenden Software den Vorzug gegenüber umfangreicher Dokumentation geben. Genauso würde ich bei einer Wüstenexpedition aber auch Wasser den Vorzug gegenüber Essen geben (*„Water over food"*). Daraus darf ich aber nicht schlussfolgern, dass ich prinzipiell auf Essen verzichten kann.

Vielmehr ist es doch so, dass in der agilen Softwareentwicklung die **Kommunikation** im Zentrum steht: Kommunikation mit den Stakeholdern, Kommunikation innerhalb der Teams, Kommunikation zwischen den Teams und Verbreitung (also Kommunikation) von Wissen. Und Dokumentation unterstützt diese Kommunikation [Zör22]. Darum lautet die Dachzeile dieses Artikels auch „Konsistent und quellcodenah *kommunizieren*". Wer dokumentiert, der kommuniziert – und vice versa.

Grundlegende Syntax

Javadocs werden in Form von **Dokumentationskommentaren** (*doc comments*) direkt *vor* den jeweiligen Modulen, Packages, Klassen, Interfaces, Konstruktoren, Methoden, Enum-Werten und Attributen platziert. Diese Block-Kommentare werden mit `/**` eingeleitet. Dabei beachte man die zwei Asteriske („Sternchen"); ein

Dokumentations-Tag	Beschreibung
	Allgemeine Block-Tags
@deprecated text	Markiert das dokumentierte Element als veraltet. Sollte immer zusammen mit der `@Deprecated`-Quellcode-Annotation (seit Java 5) verwendet werden. Der Text sollte auf einen Ersatz verweisen.
@hidden	Entfernt das dokumentierte Elemente selektiv aus der generierten HTML-Dokumentation (seit Java 9).
@param parameter description	Beschreibt den entsprechenden Konstruktor- oder Methoden-Parameter.
@return description	Beschreibt den Rückgabewert der dokumentierten Methode.
@since text	Gibt an, seit welcher Version das dokumentierte Element existiert.
@throws exception description	Beschreibt die entsprechende Exception. Siehe hierzu eigenen Abschnitt.
	Allgemeine Inline-Tags
{**@code** text}	Erlaubt das Einfügen kurzer, einzeiliger Code-Abschnitte (seit Java 5). Verhält sich äquivalent zu `<code>{@literal text}</code>`. Spitze Klammern (<, >) im Text werden dabei *nicht* als HTML-Entitäten interpretiert. Siehe hierzu auch eigenen Abschnitt.
{**@index** word/phrase description}	Fügt ein Wort oder einen Satz (in Anführungszeichen) manuell dem generierten Index hinzu (seit Java 9).
{**@literal** text}	Wie `@code`, einfach ohne Code-Formatierung (seit Java 5).
{**@summary** text}	Verwendet den darin enthaltenen Text als Zusammenfassung in der generierten Dokumentation (seit Java 10). Muss am Anfang der Main-Description stehen.
{**@systemProperty** name}	Kennzeichnet eine System-Property.
	Vererbung
{**@inheritDoc**}	Kopiert den entsprechenden Dokumentationskommentar von der geerbten Klasse beziehungsweise dem geerbten Interface. Siehe hierzu eigenen Abschnitt.
	Code verlinken und einbetten
{**@link** reference label}	Erstellt einen Link zu `reference` mit der Bezeichnung `label`. Dabei hat `reference` die allgemeine Form `package.class#member`.
{**@linkplain** reference label}	Wie `@link`, einfach ohne Code-Formatierung.
@see reference label	Erstellt einen Link zu `reference` mit der Bezeichnung `label` im Abschnitt *„See also“* der generierten HTML-Dokumenation.
@snippet	Leitet ein Code-Snippet ein (seit Java 18). Siehe hierzu [CommentSpec] und [SnippetsGuide] für weitere Details.

Tabelle 1: Ausgewählte Block- und Inline-Tags

Block-Kommentar beginnend mit `/**` ist *immer* ein Javadoc-Kommentar (wenn auch möglicherweise ein ungültiger, sofern falsch platziert), wohingegen gewöhnliche Block-Kommentare mit `/*` (also mit nur *einem* Asterisk) eingeleitet werden. Die darauffolgenden Kommentarzeilen müssen nicht zwingend mit `*` beginnen, wobei ich selten bis nie gesehen habe, dass sie ausgelassen werden. Sogar die Javadoc-Generatorfunktionen gängiger IDEs fügen die Asteriske an den Zeilenanfängen automatisch hinzu.

Ein Javadoc-Kommentar setzt sich aus **zwei Teilen** zusammen: einer Hauptbeschreibung (*main description*), in deren Inhaltsgestaltung die Entwickler prinzipiell frei sind, und einem darauffolgenden Bereich mit Block-Tags, mit denen Parameter, Rückgabewerte, Exceptions und diverse weitere Eigenschaften in einer systematischen Schreibweise beschrieben werden (siehe folgender Abschnitt). Die Main-Description kann theoretisch ausgelassen werden (allerdings nicht empfohlen), von den Block-Tags müssen hingegen alle notwendigen aufgeführt werden, ansonsten kommt es beim Generieren der Dokumentation zu einer Warnung oder gar zu einem Fehler.

Mit dem **Kommando** `javadoc`, welches Teil des JDK ist, wird aus dem Quellcode und den dazugehörigen Javadoc-Kommentaren eine gut strukturierte und navigierbare HTML-5-Dokumentation erzeugt. Mittels *Doclets* (von Drittherstellern – oder selbst programmiert) ließen sich neben HTML auch andere Ausgabeformate erzeugen, wobei ich persönlich keinen großen Sinn darin sehe, auf die seit sehr vielen Jahren etablierte HTML-Variante zu verzichten.

Der `javadoc`-Generator stellt eine Fülle an **Optionen** in Form von Aufrufparametern zur Verfügung, die sich allerdings meist nur auf Details und Spezialfälle beziehen und daher in diesem Artikel nicht weiter behandelt werden. Kommt hinzu, dass die klassischen Java-IDEs wie IntelliJ IDEA oder Eclipse bereits Dialogfelder zum Generieren der Javadoc-API-Dokumentation zur Verfügung stellen, in denen die gängigen Optionen gesetzt werden können. Wer sich darüber hinaus einen Überblick über alle Optionen verschaffen will, sei auf [javadocCommand] verwiesen.

Der erste Satz der Main-Description hat eine besondere Bedeutung: Er dient jeweils als **zusammenfassende Beschreibung** (engl. *summary description*) innerhalb der generierten Klassen-, Konstruktoren-, Methoden- und anderer Übersichten. Er sollte daher stets mit entsprechender Sorgfalt formuliert werden. Ein klassischer Fehler, der dabei passieren kann, ist der versehentliche vorzeitige Abbruch des Satzes aufgrund eines Punkts und eines darauffolgenden Leerraums (engl. *whitespace*), wie zum Beispiel bei Abkürzungen (`z.B. `, `etc. `) oder bei Ordnungszahlen (`1. `, `2. `). Um derartige Probleme zu vermeiden, empfiehlt es sich, die kritischen Stellen mithilfe des *Inline-Tags* `{@literal}` zu kodieren (hier also: `{@`

literal z.B.}) oder gleich – wie seit Java 10 möglich – die gesamte Summary-Description in {@summary text} zu verpacken. Ich finde die neue {@summary}-Variante ausgesprochen schön und werde daher ausschließlich diese in meinen Beispielen verwenden.

Alle **darauffolgenden Absätze der Main-Description** sollten mit HTML-<p>-Tags versehen werden. Allgemein erlauben Javadoc-Kommentare die Einbettung von HTML. Gleichzeitig sollte der Grundsatz befolgt werden, dass Doc-Comments sowohl im Quellcode als auch in der generierten Dokumentation ähnlich gut lesbar sein sollten, was gegen einen exzessiven Gebrauch ausgefallener HTML-Formatierung spricht.

Ausgewählte Block- und Inline-Tags

Javadoc kennt zwei Arten von Tags: *Block-Tags* der Form @identifier content, die jeweils für sich selbst am Zeilenanfang stehen müssen, und *Inline-Tags* der Form {@identifier content} (also mit geschweiften Klammern), die innerhalb anderer Beschreibungen verwendet werden dürfen.

Tabelle 1 zeigt einige ausgewählte Block- und Inline-Tags. Für eine vollständige und sehr gute Beschreibung sämtlicher Tags sei an die offizielle *Documentation Comment Specification for the Standard Doclet (JDK 19)* verwiesen [CommentSpec]. In der Tabelle weggelassen habe ich vor allem Tags betreffend Serialisierung und Modulsystem (seit Java 11). Ebenso weggelassen habe ich @author und @version, weil Quellcode heute in der Regel von mehreren Personen geschrieben wird, und das Nachführen von Versionsnummern in jeder Klasse durch die Verwendung von Versionsverwaltungssystemen wie Git heute praktisch überflüssig geworden ist. @author und @version werden vom Javadoc-Generator mittlerweile nur noch beim Setzen entsprechender Aufrufparameter übernommen und sind standardmäßig deaktiviert [javadocCommand].

Listing 1 zeigt einige der erwähnten Block- und Inline-Tags in Form eines umfassenden Beispiels.

Exceptions

Java unterscheidet bekanntlich zwischen geprüften Ausnahmen (engl. *checked exceptions*) und ungeprüften Ausnahmen (engl. *unchecked exceptions*). Checked Exceptions *müssen* abgefangen, behandelt oder weitergeworfen werden, wohingegen unchecked Exceptions im Code – salopp gesagt – ignoriert werden können.

Exceptions werden in Javadoc mittels @throws-Tags beschrieben. @exception ist ein Synonym dazu, sollte aber nicht (mehr) verwendet werden. Mit einer vollständigen Dokumentation sämtlicher Aufrufparameter (via @param) und Exceptions (via @throws) können die Vorbedingungen (engl. *preconditions*) eines Konstruktors beziehungsweise einer Methode – und damit der „Vertrag" zwischen Aufrufer und Aufgerufenem – effektiv spezifiziert werden.

In der throws-Klausel einer Konstruktor- und Methodensignatur sollten nach Konvention *nur* checked Exceptions deklariert werden. Im dazugehörigen Javadoc sollten hingegen *auch* die unchecked Exceptions mittels @throws dokumentiert werden, sofern sie im Verantwortungsbereich des Aufrufers liegen (was typischerweise bei NullPointerExceptions und IllegalArgumentExceptions entsprechender Parameter der Fall sein dürfte). Diese Konvention hilft Entwicklern auch, auf einen Blick zu erkennen, ob es sich bei einer Exception um eine geprüfte Ausnahme (sowohl in throws als auch in @throws aufgeführt) oder um eine ungeprüfte Ausnahme (nur in @throws aufgeführt) handelt [BloItem74]. Listing 1 zeigt auch die beispielhafte Verwendung der @throws-Tags.

Vererbung

Javadoc-Kommentare können an Unterklassen oder -Interfaces vererbt werden. Überschreibende oder implementierende Methoden erhalten vom Javadoc-Generator automatisch einen Link auf die übergeordnete Methode („Obermethode"), verbunden mit der Bemerkung *„Overrides"* beziehungsweise *„Specified by"*.

```
import java.net.ConnectException;
import java.time.Duration;
import java.time.LocalTime;

/**
 * {@summary Demonstrates the use of the most common Javadoc block and
 * inline tags.}
 *
 * <p>The main description can consist of several paragraphs.
 *
 * <p>The first sentence of the main description represents the summary.
 * To avoid any confusion, it is recommended to use the new {@code
 * {@summary}} inline tag, which is availabe since Java 10.
 *
 * <p>Doc comments can also contain <em>HTML code</em>. If a text snippet
 * also contains <em>HTML entities</em>, the easiest way to mask these is
 * by using the {@code {@literal}} inline tag, which allows mathematical
 * statements with a {@index "relational operator" An operator with angle
 * brackets that might easily be confused with HTML entities.} like
 * {@literal 3 < 5}.
 *
 * @since 2022.10
 */
public class BlockAndInlineTags {

  /**
   * The constant URL string to the Swiss NTP (network time protocol)
   * server pool.
   */
  public static final String NTP_SERVER_URL_STRING = "ch.pool.ntp.org";

  /**
   * {@summary Creates the example class.}
   *
   * <p>For the sake of consistency, no-arg constructors should be
   * explicitly implemented because automatically generated default
   * constructors cannot be documented with Javadoc.
   */
  public BlockAndInlineTags() {
    super();
  }

  /**
   * {@summary Returns the duration between the given time and the
current
   * time.}
   *
   * @param time the time to which the difference from the current time
   *     should be calculated
   * @return the {@linkplain java.time.Duration duration} between {@code
   *     time} and the current time. If the given time is <em>later</em>
   *     than the current time, then the duration will be
   *     <em>positive</em>. If the given time is <em>earlier</em> than the
   *     current time, then the duration will be <em>negative</em>.
   * @throws ConnectException if the connection to the NTP server cannot
   *     be established
   * @throws NullPointerException if {@code time} is {@code null}
   */
  public Duration calculateDifferenceToCurrentTime(LocalTime time)
      throws ConnectException {

    // Omit the implementation and simply throw an exception.
    throw new ConnectException("Exception Code GTN9");
  }
}
```

Listing 1: Demonstration verschiedener Javadoc-Aspekte

```
import java.net.ConnectException;
import java.time.Duration;
import java.time.LocalTime;

/**
 * {@summary Demonstrates doc comment inheritance and the use of the
 * {@code @inheritDoc} tag.}
 *
 * @since 2022.10
 */
public class InheritanceTags extends BlockAndInlineTags {

  /** {@summary Creates the example subclass.} */
  public InheritanceTags() {
    super();
  }

  /**
   * {@inheritDoc}
   *
   * <p>But here the main description adds another sentence.
   *
   * @return {@code null}, just to demonstrate the use of a different
   * {@code @return} tag
   */
  @Override
  public Duration calculateDifferenceToCurrentTime(LocalTime time)
      throws ConnectException {

    // Do nothing.
    return null;
  }
}
```

Listing 2: Vererbung bei Javadoc

Fehlende Main-Descriptions oder fehlende `@param`-, `@return`- oder `@throws`-Tags überschreibender oder implementierender Methoden werden automatisch von den entsprechenden Obermethoden kopiert. Bei `@throws` gilt das aber nur für *deklarierte* Exceptions, was nach dem im vorherigen Abschnitt Gesagten in der Regel unchecked Exceptions ausschließt.

Mit dem `{@inheritDoc}`-Inline-Tag lässt sich an genau dieser Stelle die entsprechende Main-Description oder der entsprechende `@param`-, `@return`- oder `@throws`-Kommentar der Obermethode einfügen. In der Regel wird um das `{@inheritDoc}` noch Text herum geschrieben, sodass der entsprechende Kommentar der Obermethode buchstäblich eingebettet wird.

Besagte Regeln gelten aber nur für Methodenkommentare, nicht für Konstruktoren, Attribute oder innere Klassen. In [CommentSpec] werden die Vererbungsregeln noch etwas detaillierter beschrieben. Listing 2 zeigt ein kurzes Beispiel als Unterklasse der Klasse aus Listing 1. Abbildung 1 zeigt einen Screenshot des dazugehörigen generierten Javadoc-HTMLs.

Sprachliches

Javadoc-Kommentare als Schnittstellenbeschreibungen sollten bevorzugt in englischer Sprache verfasst werden, da (neu) angestellte Softwareentwickler zunehmend internationaler werden. Allerdings würde ich diese Regel nicht so starr umsetzen. Ein lokal verwurzelter mittelständischer Metallbaubetrieb mit Entwicklungsabteilung mag noch gute Gründe haben, bevorzugt auf Deutsch zu dokumentieren, im Gegensatz zu einer international ausgerichteten Bank oder Versicherung. Vor allem wenn es innerhalb der Software um Fachlogik geht, die sich nur umständlich in andere Sprachen übersetzen lässt, empfehle ich, dem Grundsatz zu folgen: lieber gutes Deutsch als schlechtes Englisch. Denn mit Letzterem ist wirklich niemandem geholfen, weder dem Verfasser noch dem Leser.

Wer sich für Englisch entscheidet, sollte sich bewusst sein, dass amerikanisches Englisch und britisches Englisch prinzipiell zwei verschiedene Sprachen sind. Es empfiehlt sich – falls noch nicht geschehen – möglichst frühzeitig eine firmenweite Richtlinie anzustreben, welche Varietät der englischen Sprache denn nun hausintern und nach außen verwendet werden soll. Erfahrungsgemäß handelt es sich dabei nämlich nicht um ein unbedeutendes Detail, sondern um einen grundlegenden und weitreichenden Entscheid.

Nach Konvention sollte die Beschreibung eines `@param`- und `@return`-**Tags** einer **Nominalphrase** (engl. *noun phrase*) entsprechen. Dabei hat der „Satz“ (ohne Verb) einen nominalen Kopf, beginnt prinzipiell mit Kleinschreibung (außer das Nomen selbst wird großgeschrieben) und endet *ohne* Punkt, außer wenn danach noch weitere Sätze folgen.

Die Beschreibung eines `@throws`-Tags enthält zu Beginn das kleingeschriebene Wort „if“ beziehungsweise „wenn“ und beschreibt anschließend – mit Verb –, unter welchen Umständen besagte **Exception** auftreten kann. Auch hier wird *nicht* mit einem Punkt abgeschlossen, sofern danach nichts mehr folgt.

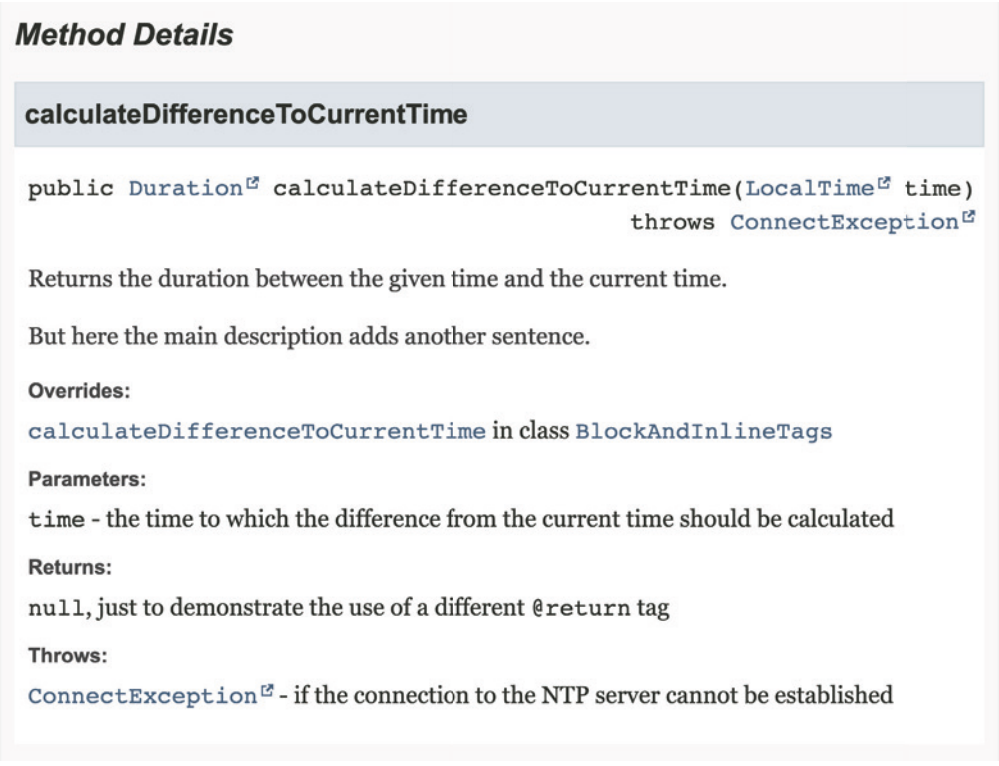


Abb. 1: Screenshot der generierten HTML-Dokumentation

Der erste und damit zusammenfassende Satz (der streng genommen gar kein ganzer Satz ist) einer **Klassen-, Interface- und Attributbeschreibung** sollte ebenfalls eine Nominalphrase sein, dieses Mal aber mit Großschreibung am Anfang und abschließendem Punkt am Ende. Letzteres hat auch technische Gründe, wie weiter oben im Detail ausgeführt.

Die zusammenfassende Beschreibung eines **Konstruktors** und einer **Methode** sollte hingegen stets eine Verbalphrase (engl. *verb phrase*) sein, das heißt ein „Satz" mit Verb, aber ohne Substantiv. Auch hier wird mit Großschreibung begonnen und mit einem Punkt abgeschlossen.

Alle erwähnten (Teil-)Sätze sollten stets in der **dritten Person Singular** formuliert werden (zum Beispiel *„Returns the duration […]."* oder *„Gibt die Dauer […] zurück."*) und *nicht* als Imperativ (Befehlsform, zum Beispiel *„Return the duration […]."* oder *„Gib die Dauer […] zurück."*). Letzteres sollte reinen Quellcode-Kommentaren vorbehalten bleiben. Dort wird „befohlen", was der darauffolgende Codeabschnitt tun soll.

Alle diese kleinen, aber entscheidenden sprachlichen Feinheiten sind in [BloItem56] beschrieben und in Listing 1 noch mal ersichtlich.

Code verlinken und einbetten

Mit den Inline-Tags `{@link reference label}` und `{@linkplain reference label}` lassen sich innerhalb von Beschreibungen Links auf andere Typen und Methoden einfügen. Die beiden Inline-Tags unterscheiden sich nur in der resultierenden Formatierung: `{@link}` stellt den Link in Code-Formatierung dar, `{@linkplain}` in normaler Schrift.

Die `reference`, die analog dem `href`-Attribut eines HTML-`<a>`Tags entspricht, folgt dabei dem allgemeinen Schema `package.class#member`. Ein Verweis auf die `equals`-Methode der Klasse `String` mit dem Label *„equals"* sähe damit zum Beispiel wie folgt aus:

```
See {@link String#equals(Object) equals} for more details.
```

Obiger Kommentar könnte auch mittels Block-Tag `@see` realisiert werden. In der generierten HTML-Dokumentation erscheint es dann in einem separaten Abschnitt mit der Überschrift *„See also"*:

```
@see String#equals(Object) equals
```

Ganz kurze, einzeilige Code-Abschnitte lassen sich bevorzugt mit `{@code text}` einbinden. HTML-Entitäten wie spitze Klammern werden dabei ignoriert. Einzig bei geschweiften Klammern ist darauf zu achten, dass die Anzahl öffnender Klammern im Code auch derjenigen der schließenden Klammern entspricht. `{@literal text}` ist das Gegenstück zu `{@code}`, einfach ohne Code-Formatierung.

Für längere Code-Abschnitte musste früher auf präformatierte HTML-Blöcke der Form `<pre>{@code ...}</pre>` zurückgegriffen werden. Seit Java 18 – also noch ganz frisch – gibt es mit der Umsetzung von JEP 413 ein Block-Tag `@snippet` [JEP413]. Dieser erlaubt sowohl das direkte Einfügen als auch das externe Verlinken von Code-Snippets, eine einstellbare Einrückung, Hervorhebungen innerhalb des Quellcodes, automatische Textersetzungen (auch mit regulären Ausdrücken), die Spezifizierung einzelner Regionen und einiges mehr. Eine Abhandlung des umfangreichen Funktionsumfanges würde den hier vorliegenden Artikel sprengen. Der Leser sei daher auf [SnippetGuide] für eine umfassende Einführung mit Beispielen verwiesen.

Fazit

Es ist wahrscheinlich wenig zielführend, jeden Softwareentwickler im Haus gleichermaßen zu Dokumentationsarbeiten zu „zwingen". Jede Person hat ihre fachlichen Stärken und Schwächen. Eine nur widerwillig oder minderwertig erstellte Dokumentation wird auch für die Leser von wenig Nutzen sein.

Dennoch kann effektives Dokumentieren in meinen Augen genauso gelernt werden wie Programmieren. Es braucht nur den Willen, ein Konzept und einen objektiven Maßstab („Lehrmeister" oder Buch), an dem man sich orientieren kann. Es erstaunt doch immer wieder, dass sich Entwickler und POs zwar in Sachen Datenbanken, agilen Methoden, den Neuerungen in Java 17 (LTS) oder dem Einsatz der Amazon Cloud (AWS) weiterbilden, dem Thema Dokumentation aber oft nur mit Improvisation oder Schulterzucken begegnen.

Ich möchte deswegen gerne einen Lösungsvorschlag präsentieren, den ich in meinen früheren Artikeln in der ein oder anderen Form schon einmal vorgestellt habe. Unterm Strich ist es wahrscheinlich effizienter, wenn es innerhalb des Entwicklerstamms ein paar ausgewiesene Experten gibt, die sich jeweils einer bestimmten Thematik (hier: der Dokumentation) annehmen. Wer gerne und gut dokumentiert, wird der gesamten Entwicklungsabteilung und damit natürlich auch der Firma einen großen Gefallen tun. So oder so sollte der Punkt *„Javadoc erstellt respektive nachgeführt"* ein fester Bestandteil bei Code-Reviews sein.

Stefan Zörner geht in [Zör22] sogar noch einen Schritt weiter und schlägt die Rolle eines „Doctators" vor, der unter anderem einen Überblick über die Dokumentation hat und schafft, den Prozess des Dokumentierens sowohl in den menschlichen als auch technischen Ablauf (Build-Pipeline) integriert, die Teams beim Dokumentieren unterstützt und die Aktualität und Konsistenz der Inhalte überwacht. Wenn sich hausintern keiner findet, wäre es auch eine denkbare Option, für diese Rolle anfangs eine externe Person beizuziehen.

Ich hoffe, mit vorliegendem Artikel hinsichtlich der Dokumentationsthematik im Allgemeinen sensibilisiert und hinsichtlich Javadoc im Speziellen zu mehr Hintergrundwissen, Detailkenntnis und damit gesteigerter Konsistenz und Qualität beigetragen zu haben.

Literatur und Links

[BloItem56] J. Bloch, Effective Java, Item 56: Write doc comments for all exposed API elements, 3rd Edition, Addison-Wesley, 2018

[BloItem74] J. Bloch, Effective Java, Item 74: Document all exceptions thrown by each method, 3rd Edition, Addison-Wesley, 2018

[CommentSpec] Documentation Comment Specification for the Standard Doclet (JDK19), https://docs.oracle.com/en/java/javase/19/docs/specs/javadoc/doc-comment-spec.html

[javadocCommand] The javadoc Command, https://docs.oracle.com/en/java/javase/19/docs/specs/man/javadoc.html

[JEP413] OpenJDK, JEP 413: Code Snippets in Java API Documentation, https://openjdk.java.net/jeps/413

[Lil20] C. Lilienthal, Langlebige Software-Architekturen, 3. Auflage, dpunkt.verlag, 2020

[Manifesto] Manifesto for Agile Software Development, https://agilemanifesto.org

[SnippetsGuide] Programmer's Guide to Snippets, https://docs.oracle.com/en/java/javase/18/code-snippet/index.html

[Ull22] C. Ullenboom, Java ist auch eine Insel, 16. Auflage, Rheinwerk Verlag, 2022

[Zör22] S. Zörner, Softwarearchitekturen dokumentieren und kommunizieren, 3. Auflage, Hanser, 2022