Fachbereich 3 - Informatik

Javadoc-Erkennung mittels Detektor (Javadoc Detection)

Bachelorarbeit
im Studiengang
Informatik

von
Michel Krause

abgegeben am:
11. November 2019

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Dr. Hui Shi

Betreuer: Marcel Steinbeck

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderen Prüfungsgremium vorgelegen.

Datum: _____ Unterschrift: _____

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1. Einleitung

1.1. Motivation

Aufgrund der immer weiter fortschreitenden Entwicklung der Technologie und den unzähligen neuen Einsatzgebieten werden Softwaresysteme immer komplexer. Da Softwaresysteme oftmals untereinander vernetzt und mit Hardware verknüpft sind, ist es unabdingbar, dass diese Softwaresysteme für Änderungen anpassbar, erweiterbar und wiederverwendbar sind. Denn es wäre nicht effizient für jede Änderung die Software komplett neu schreiben zu müssen.
In dem ISO-Standard 25010 werden einige Qualitätskriterien für Softwaresysteme festgelegt. Diese Qualitätskriterien sind dabei ein maßgeblicher Indikator für die Qualität von Software. Ein wichtiges Kriterium für die eben beschriebene Anpassbarkeit und Erweiterbarkeit, ist die Wartbarkeit. Die Wartbarkeit definiert sich über die Modularität, die Wiederverwendbarkeit, die Analysierbarkeit, die Änderbarkeit und die Testbarkeit. [Gno16]
Damit ein Softwaresystem effektiv und vor allem effizient modifiziert beziehungsweise gewartet werden kann, ist es wichtig, dass der Quellcode des Systems nachvollziehbar ist. Nachvollziehbar ist ein Quellcode, sobald dieser selbsterklärend geschrieben oder gut dokumentiert ist. [Mei09]
Dennoch wird die Dokumentation von Software häufig ganz vernachlässigt oder zumindest unzureichend durchgeführt. [Ull12, S. 1259]
Einen Ansatz zur Verbesserung der Qualität von Software bietet die Refaktorisierung. Die Refaktorisierung versucht mit sogenannten Code Smells auf schlechte Programmierpraktiken hinzuweisen und die Software zu verbessern. Dies erreicht die Refaktorisierung dadurch, dass die Codekomplexität durch eine Neustrukturierung des Codes reduziert wird, ohne jedoch die Funktionalität des Codes zu verändern. Dadurch kann der Code später schneller, leichter und kostengünstiger gewartet werden. [BH14]
In dem gleichnamigen Buch “Refactoring“ von Martin Fowler, werden viele verschiedene solcher Code Smells beziehungsweise Bad Smells erläutert. Martin Fowler bietet mit dem Buch einen Leitfaden für professionelle Programmierer zur Durchführung einer Refaktorisierung. [FBB+99, S. 9]

Die Software AG der Universität Bremen bietet eine öffentliche Java Bibliothek mit dem Namen LibVCS4j[1] an. Diese Bibliothek stellt verschiedene Module zur Analyse von Softwaresystemen bereit. Mithilfe dieser Module ist die Bibliothek bereits in der Lage folgende Code Smells zu erkennen:

- God Class[2]
- Long Method[3]
- Long Parameter List[4]
- Cycle, wird zur Zyklenerkennung benötigt
- Method Chain[5]
- Comments[6]
- Data Class[7]
- Unused Code[8]
- Switch Statements[9]
- Temporary Field[10]

Mittels der Bibliothek lassen sich nicht nur Code Smells analysieren, sie bietet darüber hinaus auch weitere Analysen für Software-Repositories. Jedoch fehlt der LibVCS4j-Bibliothek bisher ein Javadoc-Detektor, der analysieren kann, ob und mit welcher Güte ein Java-Quellcode dokumentiert wurde.
Das Ziel dieser Bachelorarbeit besteht darin, die LibVCS4j-Bibliothek so weiterzuentwickeln, dass eine Javadoc-Dokumentation auf ihre Vollständigkeit und Korrektheit hin analysiert werden kann. Dabei meint "Vollständigkeit", dass alle vorhanden Elemente des Java-Codes, die dokumentiert werden müssen, dokumentiert sind und ihre vorgegebene Mindestlänge einhalten. Mit der Korrektheit ist hier gemeint, dass die Dokumentation keine überflüssigen beziehungsweise falschen Informationen enthält. Dies ist zum Beispiel dann der Fall, wenn der Javadoc-Kommentar, aus dem die Dokumentation generiert wird, einen Parameter beschreibt der im dazugehörigen Quellcode nicht existiert. Diese Anforderungen setzen voraus, dass der Javadoc-Detektor hoch konfigurierbar sein muss.

[1] https://github.com/uni-bremen-agst/libvcs4j
[2] https://en.wikipedia.org/wiki/God_object
[3] https://sourcemaking.com/refactoring/smells/long-method
[4] https://sourcemaking.com/refactoring/smells/long-parameter-list
[5] https://sourcemaking.com/refactoring/smells/message-chains
[6] https://sourcemaking.com/refactoring/smells/comments
[7] https://sourcemaking.com/refactoring/smells/data-class
[8] https://refactoring.guru/smells/dead-code
[9] https://sourcemaking.com/refactoring/smells/switch-statements
[10] https://sourcemaking.com/refactoring/smells/temporary-field

1.2. Forschungsfragen

Der Begriff Javadocables, der in den Forschungsfragen vorkommt, umfasst die Elemente, die mit einem Javadoc-Kommentar versehen sein können. Eine kleine Erklärung zu diesem Begriff wird in dem zweiten Kapitel unter dem Abschnitt Javadocables gegeben.

RQ1: Wie entwickeln sich die Javadoc Code Smells in den Projekten im Laufe der Zeit?
Mithilfe dieser Forschungsfrage soll untersucht werden, wie sich die analysierten Projekte über ihre Historie hinweg in Hinblick auf ihre Javadoc Code Smells entwickeln. Dafür werden von den analysierten Projekten verschiedene Daten zu jeder ihrer Revisionen erhoben. Eine Revision stellt den Stand einer Software zu einem bestimmten Zeitpunkt dar. Danach wird mittels der Mann-Kendall-Trendanalyse überprüft, welche Projekte einen aufsteigenden, absteigenden oder keinen Code Smell Trend besitzen. Dabei wird gezielt auf Auffälligkeiten eingegangen. Eine solche Auffälligkeit wäre zum Beispiel ein Projekt, das einen aufsteigenden Trend für die Javadoc Code Smells insgesamt und dennoch einen absteigenden Trend für den Javadoc Code Smell *"no Javadoc"* besitzt. Der absteigende Trend für *"no Javadoc"* bedeutet hierbei, dass dieses Projekt im Laufe der Zeit mehr Javadocables mittels eines Javadoc-Kommentars dokumentiert hat, bei denen dies bisher komplett gefehlt hat. Des Weiteren wird analysiert wie sich die Trends über die Anzahl der Javadocables zu den Trends der insgesamten Javadoc Code Smells verhalten.

RQ2: Wie häufig kommt es in den Projekten zu Javadoc Verstößen und wie sind diese Verstöße auf die Javadocables verteilt?
Diese Forschungsfrage soll beantworten, wie häufig es in den analysierten Projekten zu Javadoc Verstößen und somit zu Javadoc Code Smells kommt. Außerdem soll untersucht werden, welche von diesen Javadoc Code Smells am häufigsten vertreten sind. Des Weiteren soll spezifischer geschaut werden, wie sich die Javadoc Code Smells auf die jeweiligen Javadocables verteilen. Dabei werden die jeweiligen Javadoc Code Smells jedoch nicht wie in der ersten Forschungsfrage zu Oberkategorien zusammengefasst.

RQ3: Wie viele der Javadocables, die in den untersuchten Projekten zu finden sind, sind von Javadoc Code Smells betroffen und wie viele sind frei von solchen Smells?
Diese Forschungsfrage dient dazu Aussagen über die Verteilung zwischen dem Befall und nicht Befall der Javadocables treffen zu können. Dabei soll untersucht werden, wie viele der Javadocables aus den analysierten Projekten von Javadoc Code Smells befallen oder unbefallen sind. Anschließend wird verglichen, welcher Javadocable am häufigsten und welcher am wenigsten befallen und frei von Javadoc Code Smells sind.

1.3. Aufbau der Arbeit

Zuzüglich zu diesem Kapitel bietet diese Bachelorarbeit noch vier weitere Kapitel. In dem nachfolgenden zweiten Kapitel werden die Grundlagen zu dem Thema dieser Arbeit erläutert. Diese Grundlagen umfassen unter anderem das Thema Java Kommentare, insbesondere des Javadoc-Kommentars sowie die Themen Code Smells, Spoon und die LibVCS4j-Bibliothek. Darauf folgt die Vorstellung und Erklärung der Implementierung des Javadoc-Detektors, der dieser Arbeit zugrunde liegt. Anschließend beschäftigt sich das vierte Kapitel mit der Evaluation und der Beantwortung der Forschungsfragen. Zum Schluss dieser Bachelorarbeit wird ein Fazit gezogen sowie ein Ausblick auf weitere Forschungsmöglichkeiten und mögliche Erweiterungen für den Javadoc-Detektor vorgestellt.

2. Grundlagen

In diesem Kapitel werden die benötigten Grundlagen für die vorliegende Bachelorarbeit erörtert. Begonnen wird damit, dass vermittelt wird, was Java Kommentare sind, welche Arten von Java Kommentaren es gibt und deren jeweilige Funktion beschrieben. Anschließend wird in einem Unterkapitel näher auf den Java Kommentar Javadoc eingegangen und erläutert, welche Bestandteile dieser besitzt. Des Weiteren wird in diesem Unterkapitel der Begriff Javadocables erklärt. Danach werden die Grundlagen zu den Code Smells, der Spoon-Bibliothek und der LibVCS4j-Bibliothek nähergebracht.

2.1. Java Kommentare

Mit der Hilfe von Kommentaren ist es möglich, dem Leser Hinweise im Quellcode zu geben oder den Quellcode zu dokumentieren. Diese Kommentare werden von dem Compiler ignoriert und haben somit keinen Einfluss auf den Programmablauf. [Abt10]
Java bietet dem Nutzer drei verschiedene Möglichkeiten, um Quelltexte zu kommentieren [Ull12, S. 120-122]:

- **Zeilenkommentare**:
  Der Zeilenkommentar kommentiert den Rest der jeweiligen Zeile aus. Eingeleitet wird dieser durch zwei Schrägstriche "//". Alle Zeichen die hinter diesen Schrägstrichen stehen bis zum nächsten Zeilenumbruch sind Teil des Kommentars.
  Beispiel:

```
1  preisBrutto = preisNetto * umsatzsteuer; // Berechnung des Bruttopreises.
2
```

Listing 2.1: Beispiel für einen Zeilenkommentar.

- **Blockkommentare**:
  Der Blockkommentar wird durch “/*“ eingeleitet und durch “*/“ beendet. Dadurch werden alle Zeilen dazwischen auskommentiert. Dabei muss jedoch bedacht werden, dass der Kommentartext kein “*/“ beinhalten darf, da sonst der Blockkommentar beendet wird. Eine Verschachtelung von Blockkommentaren ist nicht möglich.
  Beispiel:

```
1  /*
2   * TODO: Funktion umbenennen.
3   * Die Funktion benötigt einen eindeutigen Namen, mit dessen Hilfe ersichtlich wird,
4   * welche Aufgaben diese Funktion übernimmt.
5   */
6  doSomethingWeirdWithText(text);
7
```

Listing 2.2: Beispiel für einen Blockkommentar.

- **Dokumentationskommentare**:
  Diese Art des Kommentars wird auch Javadoc-Kommentar oder kurz Javadoc genannt. Der Dokumentationskommentar stellt einen besonderen Blockkommentar dar, aus dem später die API-Dokumentation generiert werden kann. Diese Kommentar-Art wird durch “/**“ eingeleitet und durch “*/“ beendet. Dazwischen befindet sich der Kommentar.
  Beispiel:

```
1  /**
2   * Methode zum Zählen der Umlaute in einem Text.
3   * Zu den Umlauten gehören die Buchstaben ä, ö und ü.
4   * @param text ist der Text dessen Umlaute gezählt werden sollen.
5   * @return die Anzahl der gezählten Umlaute.
6   */
7  public int zaehleUmlaute (String text) {
8      ...
9  }
10
```

Listing 2.3: Beispiel für einen Dokumentationskommentar.

Martin Fowler benennt in seinem Buch “Refactoring - Improving the Design of Existing Code“ Kommentare als Indikatoren für Code Smells, denn an den Stellen, an denen Kommentare gehäuft vorkommen, ist schlechter Code vorhanden. Fowler sagt aus, dass dieser Code umgeschrieben werden sollte, damit dieser verständlicher ist, statt Kommentare zu verwenden. Jedoch sagt Fowler ebenfalls aus, dass Kommentare die Hinweise darüber geben, warum ein Entwickler den Code so geschrieben hat, gut sind. [FBB+99, S. 63-72]
Damit die Aussage von Fowler besser nachvollziehbar ist, ist es hilfreich die oben genannten Kommentare, wie Bill Sourour, in zwei Kategorien einzuteilen. Zum einen die Dokumentationskommentare und die Erläuterungskommentare.
Die Dokumentationskommentare sind dabei für diejenigen gedacht, die den Quellcode benutzen, aber nicht lesen. Dies ist zum Beispiel dann der Fall, wenn eine öffentliche Bibliothek oder ein Framework zur Verfügung gestellt wird, in diesen beiden Fällen wird eine API-Dokumentation benötigt, damit andere Entwickler die API nachvollziehen und benutzen können. Des Weiteren sagt Sourour aus, dass um so weiter diese Dokumentation von dem Quellcode entfernt angelegt wird, desto wahrscheinlicher ist es, dass diese veraltet, da die Entwickler oftmals vergessen, diese mit zu aktualisieren. Eine gute Möglichkeit um dies zu verhindern, bietet die Vorgehensweise, die Dokumentation direkt in den Code mit einzubetten und diese dann anschließend mit einem Tool zu extrahieren, wie es zum Beispiel mit dem Tool Javadoc bei Javadoc-Kommentaren der Fall ist. Der Nachteil bei dieser Möglichkeit ist jedoch, dass die Datei, in der nun der Quellcode inklusive Dokumentation geschrieben ist, viel größer und dadurch unübersichtlicher wird. Dafür nennt Sourour eine Lösung, und zwar unterstützen die meisten IDEs die Möglichkeit Kommentare des Codes zu reduzieren, in dem sie ausgeblendet werden, falls diese nicht benötigt werden.
Dahingegen richten sich die Erläuterungskommentare an alle, die den Code warten oder modifizieren müssen. Diese Kommentare zeigen auf, an welchen Stellen der Code zu komplex ist, da dieser Code ohne Kommentar möglicher Weise nicht verständlich ist. Wie auch Fowler, benennt Sourour diese Art von Kommentaren als einen Indikator für Code Smells. Außerdem stimmt er mit Fowler überein, dass diese Code-Stücke besser umgeschrieben werden sollten. Abschließend bezeichnet Sourour die Dokumentationskommentare als die guten und die Erläuterungskommentare als die schlechten Kommentare. [Sou17]
Rafiullah Hamedy beschreibt in seinem Artikel “A short summary of Java coding best practices“ ebenfalls, wie Sourour, zwei Arten von Kommentaren. Die Kategorien von Hamedy und Sourour unterscheiden sich dabei nur durch ihre jeweilige Benennung. Hamedy unterstreicht dabei die Wichtigkeit der Dokumentationskommentare mit einem kleinen Beispiel. In dem Beispiel geht es um die Vorstellung, dass eine Person als neuer Entwickler in die Firma kommt und sich in den vorhandenen Code einarbeiten muss, während der ursprüngliche Autor des Codes die Firma verlassen oder das Projekt gewechselt hat. [Ham18]
Viele Entwickler sind der Meinung, dass gut geschriebener Code selbsterklärend ist. [Sou17] Diese Aussage sieht Fagner Brack in seinem Artikel “Code Comment Is A Smell“ etwas kritischer, denn er schreibt, dass Entwickler auch gerne mal die selbstdokumentierende Fähigkeit ihres Codes beziehungsweise eines Erläuterungskommentars überschätzten, da sie bereits die Kenntnis darüber verfügten, wie der von ihnen geschriebene Quellcode funktionierte. Eine Lösung um dieses Problem zu verhindern, ist es, eine andere Person mit ähnlichem

Kenntnisstand beurteilen zu lassen, ob der Quellcode wirklich selbstdokumentierend ist oder nicht. [Bra16]
Damit ein Kommentar die Nachvollziehbarkeit, wie im Kapitel Motivation beschrieben, unterstützen kann, muss es sich um einen sinnvollen und fehlerfreien Dokumentationskommentar handeln.
Schließlich wird eine Software öfter gelesen als geschrieben. [Ull12, S. 1259]
Jedoch ist ein Dokumentationskommentar nicht zwangsweise ein guter Kommentar. Robert Martin beschreibt in seinem Buch "Clean Code" Fälle, in denen ein solcher Kommentar sogar schädlich ist. Einer dieser Fälle ist ein Kommentar, der veraltet und überholt ist, denn dieser verbreitet dadurch Fehlinformationen, die gegebenenfalls schädlich sein können. Ebenfalls sind ungenaue Kommentare nicht förderlich, da diese den Leser in die Irre führen. Des Weiteren sind für Kommentare ungeeignete Informationen und schlecht geschriebene Kommentare ebenfalls nicht gut. Dabei meint ungeeignete Informationen, Informationen die an anderer Stelle besser aufgehoben wären. [Mar13]
Zu der eben beschriebenen Meinung von Robert Martin gibt es ein passendes abschließendes Zitat von Ron Jeffries, welches Fagner Brack in seinem Aufsatz verwendete.[Bra16]

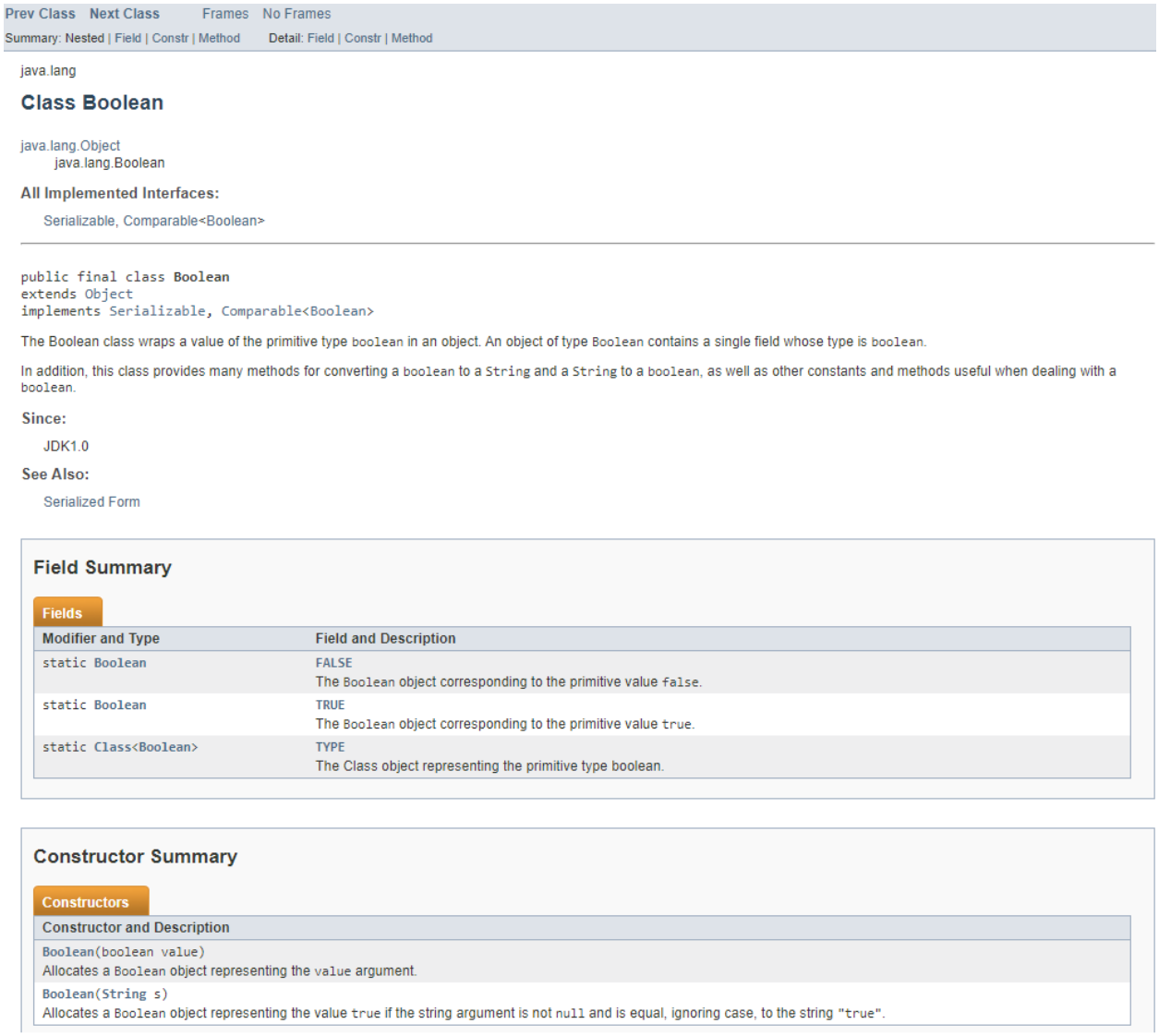
> "Code never lies, comments sometimes do." — Ron Jeffries

In dem nachfolgenden Unterkapitel werden die Javadoc-Kommentare näher erläutert.

2.1.1. Javadoc

Ein Javadoc-Kommentar steht immer unmittelbar vor der Deklaration eines Javadocables. Der Name Javadoc stammt dabei von einem im JDK (Java Development Kit) mitgelieferten Kommandozeilen-Tool namens "javadoc". Dieses Tool ist in der Lage, mit der Hilfe von den Dokumentationskommentaren Dokumentationen in Form von HTML-Seiten zu generieren. [Die01]
Dafür durchsucht das Tool den gegebenen Quelltext nach Deklarationen und extrahiert deren Dokumentation. Zu beachten ist dabei, dass das Tool standardmäßig nur öffentliche Deklaration in die Dokumentation mit aufnimmt. [Ull12, S. 1262-1263]
Um nicht öffentliche Deklarationen mit zu dokumentieren, müssen dem Tool zusätzliche Optionen übergeben werden, beispielsweise "-private" um private-Deklarationen mit dokumentiert zu bekommen. Das Tool bietet durch diese Optionen viele verschiedene Konfigurationsmöglichkeiten. Unter anderem lässt sich zum Beispiel auch der Autor des Quelltextes mittels des *@author*-Tags oder die Version mittels des *@version*-Tags für die Dokumentation berücksichtigen. [Sch10]
Im Folgenden werden Auszüge einer solchen Dokumentation, die von dem Javadoc-Tool generiert wurden, vorgestellt. Dafür wurde aus der Java API Dokumentation[1] die Klasse *"Boolean"* aus dem *"java.lang"*-Paket verwendet.

[1] https://docs.oracle.com/javase/7/docs/api/



Prev Class Next Class Frames No Frames
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class Boolean

java.lang.Object
java.lang.Boolean

All Implemented Interfaces:

Serializable, Comparable<Boolean>

```
public final class Boolean
extends Object
implements Serializable, Comparable<Boolean>
```

The Boolean class wraps a value of the primitive type `boolean` in an object. An object of type `Boolean` contains a single field whose type is `boolean`.

In addition, this class provides many methods for converting a `boolean` to a `String` and a `String` to a `boolean`, as well as other constants and methods useful when dealing with a `boolean`.

Since:

JDK1.0

See Also:

Serialized Form

Field Summary

Fields

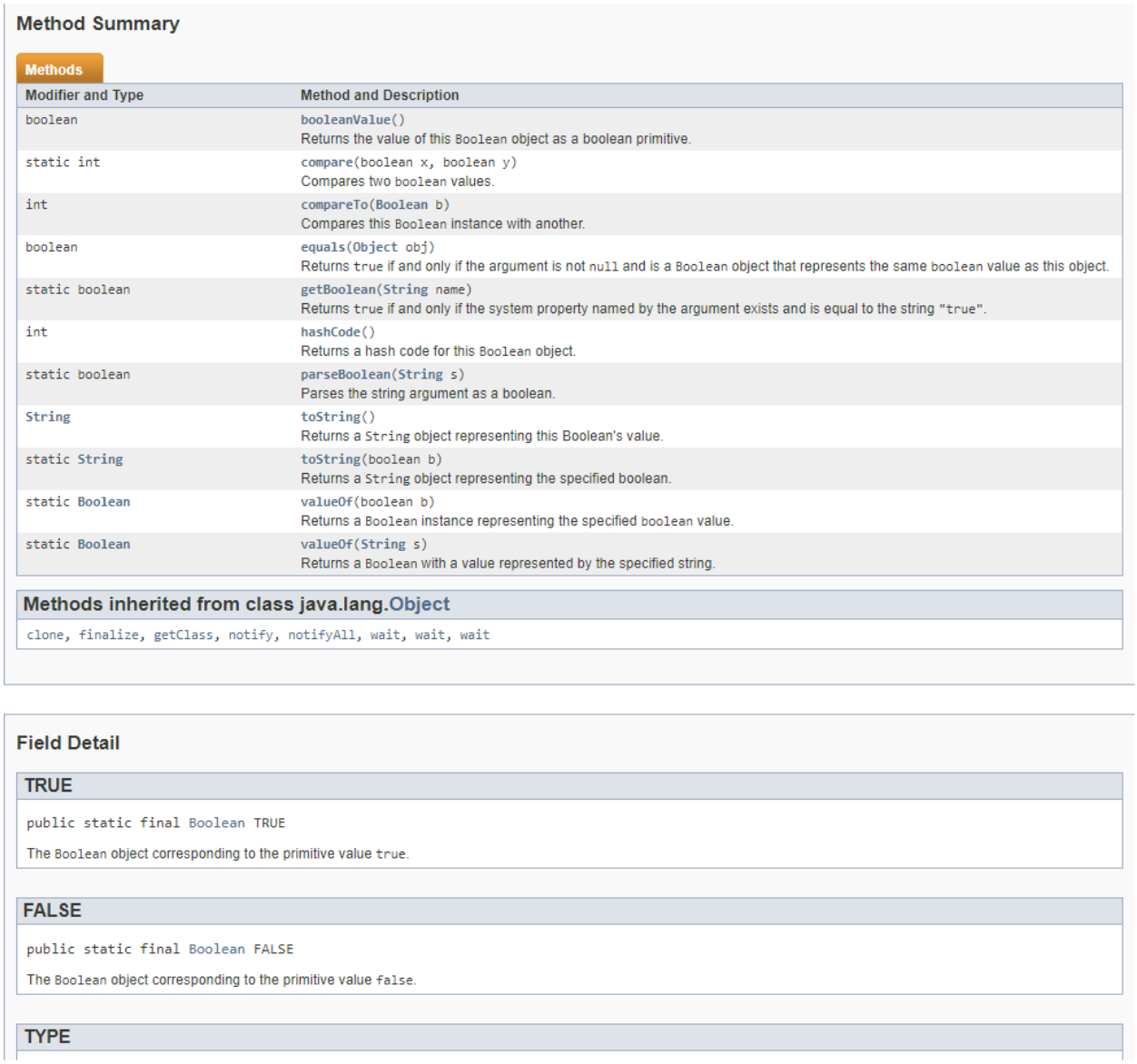
Modifier and Type	Field and Description
`static Boolean`	`FALSE` The `Boolean` object corresponding to the primitive value `false`.
`static Boolean`	`TRUE` The `Boolean` object corresponding to the primitive value `true`.
`static Class<Boolean>`	`TYPE` The Class object representing the primitive type boolean.

Constructor Summary

Constructors

Constructor and Description
`Boolean(boolean value)` Allocates a `Boolean` object representing the `value` argument.
`Boolean(String s)` Allocates a `Boolean` object representing the value `true` if the string argument is not `null` and is equal, ignoring case, to the string `"true"`.

Abbildung 2.1.: Beispiel für das Javadoc-Tool, enthält Klassenbeschreibung und Zusammenfassung der Felder und Konstruktor.



Method Summary

Methods

Modifier and Type	Method and Description
boolean	booleanValue() Returns the value of this Boolean object as a boolean primitive.
static int	compare(boolean x, boolean y) Compares two boolean values.
int	compareTo(Boolean b) Compares this Boolean instance with another.
boolean	equals(Object obj) Returns true if and only if the argument is not null and is a Boolean object that represents the same boolean value as this object.
static boolean	getBoolean(String name) Returns true if and only if the system property named by the argument exists and is equal to the string "true".
int	hashCode() Returns a hash code for this Boolean object.
static boolean	parseBoolean(String s) Parses the string argument as a boolean.
String	toString() Returns a String object representing this Boolean's value.
static String	toString(boolean b) Returns a String object representing the specified boolean.
static Boolean	valueOf(boolean b) Returns a Boolean instance representing the specified boolean value.
static Boolean	valueOf(String s) Returns a Boolean with a value represented by the specified string.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

TRUE

public static final Boolean TRUE

The Boolean object corresponding to the primitive value true.

FALSE

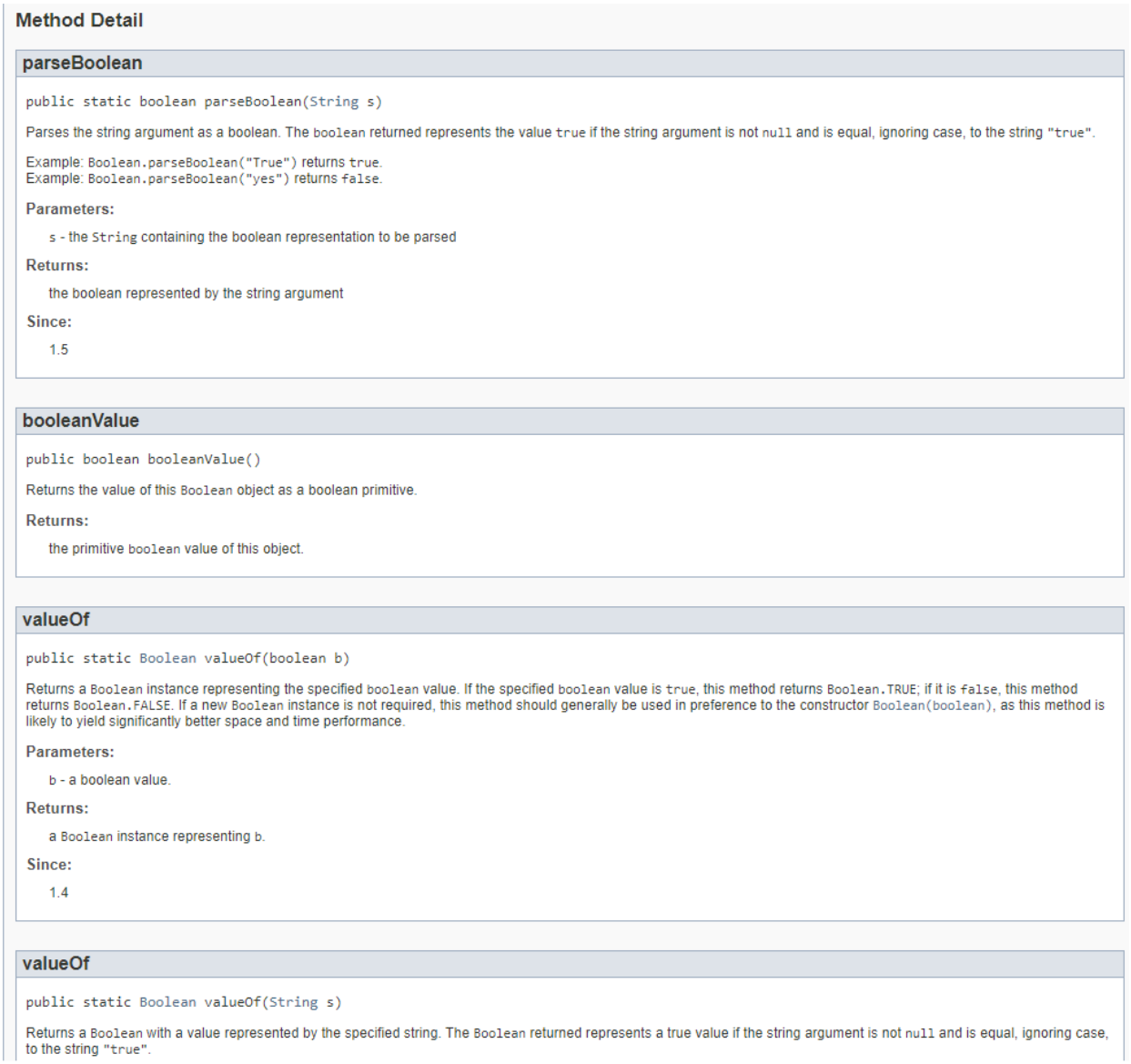
public static final Boolean FALSE

The Boolean object corresponding to the primitive value false.

TYPE

Abbildung 2.2.: Beispiel für das Javadoc-Tool, enthält die Zusammenfassung der Methoden, sowie ein Ausschnitt der Felder Details.

Die Abbildung 2.1 und Abbildung 2.2 zeigen dabei beide Zusammenfassungen der jeweiligen Inhalte der Felder, Konstruktoren und Methoden.
In der Abbildung 2.1 ist zusätzlich die Klassenbeschreibung von Boolean zu sehen und seit welcher Version diese Klasse bereitgestellt wurde.
Hingegen schneidet die Abbildung 2.2 die Beschreibung der Felder-Details an.



Method Detail

parseBoolean

`public static boolean parseBoolean(String s)`

Parses the string argument as a boolean. The `boolean` returned represents the value `true` if the string argument is not `null` and is equal, ignoring case, to the string `"true"`.

Example: `Boolean.parseBoolean("True")` returns `true`.
Example: `Boolean.parseBoolean("yes")` returns `false`.

Parameters:

`s` - the `String` containing the boolean representation to be parsed

Returns:

the boolean represented by the string argument

Since:

1.5

booleanValue

`public boolean booleanValue()`

Returns the value of this `Boolean` object as a boolean primitive.

Returns:

the primitive `boolean` value of this object.

valueOf

`public static Boolean valueOf(boolean b)`

Returns a `Boolean` instance representing the specified `boolean` value. If the specified `boolean` value is `true`, this method returns `Boolean.TRUE`; if it is `false`, this method returns `Boolean.FALSE`. If a new `Boolean` instance is not required, this method should generally be used in preference to the constructor `Boolean(boolean)`, as this method is likely to yield significantly better space and time performance.

Parameters:

`b` - a boolean value.

Returns:

a `Boolean` instance representing `b`.

Since:

1.4

valueOf

`public static Boolean valueOf(String s)`

Returns a `Boolean` with a value represented by the specified string. The `Boolean` returned represents a true value if the string argument is not `null` and is equal, ignoring case, to the string `"true"`.

Abbildung 2.3.: Beispiel für das Javadoc-Tool, enthält einen kleinen Ausschnitt der Methoden Details.

In der Abbildung 2.3 wird ein Ausschnitt der Methodendetails gezeigt. Die gezeigten Methoden wurden bereits in der Zusammenfassung in der Abbildung 2.2 vorgestellt und werden in dem Abschnitt “Method Details“ genauer erläutert.

Christian Ullenboom erklärt in seinem Buch "Java ist auch eine Insel", dass, zusätzlich zu den eben gezeigten Klassenübersichten, auch noch weitere Dokumentationen erzeugt werden. So wird beispielsweise eine Dokumentation über die Vererbung der Klassen erzeugt, um einen Überblick davon zu bekommen, welche Klassen miteinander Verwandt sind. Dies wird in der Datei *"overview-tree.html"* dokumentiert. Eine weitere zusätzliche Dokumentation bietet die Datei *"deprecated-list.html"*, in der alle veralteten Methoden und Klassen dokumentiert werden. Insgesamt beschreibt Ullenboom neun solcher zusätzlichen Dokumentationen.
[Ull12, S. 1263-1264]

Als Nächstes wird in dem folgenden Unterkapitel behandelt, wie ein Javadoc-Kommentar aufgebaut ist.

2.1.1.1. Bestandteile

Ein Javadoc-Kommentar enthält eine Beschreibung sowie gegebenenfalls sogenannte Javadoc-Tags. Für die Beschreibung und die Javadoc-Tags gibt es jeweils einen bestimmten Bereich im Javadoc-Kommentar. Die Beschreibung wird dabei in dem Kommentar ganz oben beschrieben. Im Listing 2.4 umfasst dies die Bereich (1) und (2). Die Javadoc-Tags werden unterhalb der Beschreibung beschrieben. Im Listing 2.4 umfasst dies den Bereich (3).

```
1  /**
2   * Short one line description.                          (1)
3   * <p>
4   * Longer description. If there were any, it would be    (2)
5   * here.
6   * </p>
7   * And even more explanations to follow in consecutive
8   * paragraphs separated by HTML paragraph breaks.
9   *
10  * @param  variable Description text text text.          (3)
11  * @return Description text text text.
12  */
13 public int methodName (...) {
14     // method body with a return statement
15     String test = "Hallo Welt!";
16      throw new IllegalArgumentException();
17 }
```

Listing 2.4: Ein Beispiel um die Bereiche der Javadoc-Bestandteile zu demonstrieren.

Der Quellcode des Listing 2.4 stammt von Wikipedia[2].
Im Folgenden wird spezifischer auf die Beschreibung und auf die Tags eingegangen.

[2]https://en.wikipedia.org/wiki/Javadoc

Beschreibung:
Die Beschreibung eines Javadoc-Kommentars wird in zwei Bereiche unterteilt. Zum einen die Kurzbeschreibung und zum anderen die Langbeschreibung, diese werden im späteren Verlauf der Arbeit auch als Short- und Long-Description bezeichnet. Getrennt werden diese beiden Bereiche durch den ersten Punkt im Text. Der erste Satz in der Beschreibung stellt somit die Kurzbeschreibung dar, im Listing 2.4 repräsentiert (1) diesen Satz. Die Kurzbeschreibung wird für die jeweilige Zusammenfassung, wie in der Abbildung 2.1 gezeigt wurde, verwendet. Die anschließenden Sätze bilden die Langbeschreibung, in dem Listing 2.4 umfasst dies der Abschnitt (2). Sie werden für die ausführliche Detailansicht, wie in der Abbildung 2.3 gezeigt wurde, benötigt. [ORA93b]
Des Weiteren ist es in einem Javadoc-Kommentar möglich HTML-Tags zu verwenden, um die Darstellung nach eigenen Wünschen anzupassen. Jedoch sollte auf die Überschriften-Tags (<h1>..</h1> und <h2>...</h2>) verzichtet werden, da diese laut Ullenboom zur Gliederung der Javadoc-Dokumentation verwendet werden und ihnen dadurch bestimmte Formatvorlagen zugewiesen werden. [Ull12, S. 1263]
Robert Martin zeigt in seinem Buch eine abwertende Meinung gegenüber solchen HTML-Tags in Kommentaren, er ist der Meinung, dass diese HTML-Tags das Lesen von den Kommentaren nur ungemein erschweren und deshalb nicht in den Dokumentationskommentaren verwendet werden sollten. [Mar13]

Javadoc-Tags:
Die Javadoc-Tags werden in zwei Kategorien unterteilt. Zum einen die Block-Tags und zum anderen die Inline-Tags. Die Block-Tags werden, wie eben schon erwähnt, unterhalb der Javadoc-Beschreibung dokumentiert und beschreiben dabei unter anderem Parameter, Rückgaben oder Exceptions. Dahingegen können die Inline-Tags im gesamten Javadoc-Kommentar vorkommen und setzen dort unter anderem Verweise. Außerdem werden diese Inline-Tags noch zusätzlich von geschweiften Klammern ummantelt. [Ull12]
Diese Bachelorarbeit befasst sich jedoch nur mit den Block-Tags, zum einen, weil die Spoon-Bibliothek nur diese unterstützt und zum anderen da es nicht möglich wäre, die Korrektheit solcher Verweise zu überprüfen. Im Nachfolgenden sind mit den Begriffen *“Javadoc-Tags“* oder nur *“Tags“* ausschließlich die Block-Tags gemeint.
Javadoc-Tags werden immer mit einem “@“ eingeleitet, darauf folgt dann unmittelbar ein Schlüsselwort und eine Beschreibung oder gegebenenfalls ein oder mehrere Parameter inklusive Beschreibung.
In der nachfolgenden Tabelle 2.1 werden die unterstützten Javadoc-Tags dieser Bachelorarbeit erklärt. Des Weiteren wird für jeden unterstützten Tag ein Beispiel gegeben. [ORA93b, Die01][Ull12, S. 1261-1262]

Tag inklusive Syntax	Bescheibung	Beispiel
@author name	Dieser Tag gibt den Namen des Autors an.	@author Michel Krause
@deprecated text	Dieser Tag wird in Verwendung mit der *Deprecated*-Annotation verwendet. Dabei beschreibt dieser Tag etwas Veraltetes, das nicht mehr verwendet werden sollte. Der Text erklärt dem Nutzer, was stattdessen verwendet werden soll.	@deprecated Ab Version 1.3, wurde ersetzt durch Methode x.
@exception e text	Dieser Tag ist ein Synonym für den @throws-Tag, demnach besitzt er dieselbe Funktionsweise wie @throws. Heutzutage wird der @throw-Tag bevorzugt. Für eine detaillierte Beschreibung siehe @throws.	@exception IllegalArgumentException wird geworfen, falls der übergebene Wert negativ ist.
@param p text	Mit diesem Tag werden Parameter beschrieben. Die Variable “p“ steht für den Parameternamen und die Variable “text“ ist die Beschreibung des Parameters.	@param visibility beschreibt die Sichtbarkeit der Methoden, die analysiert werden sollen.
@return text	Mittels des Tags @return wird der Rückgabewert einer Methode beschrieben.	@return gibt die aktuell gezählten Typen zurück.

@see reference	Durch diesen Tag ist es möglich auf andere Objekte, Pakete, Methoden oder Eigenschaften zu verweisen.	@see "Die Java-Programmiersprache" @see Foo bar @see String#concat(String) concat
@serial text | include | exclude	Serial dient zur Beschreibung eines standardmäßigen serialisierbares Feld. Die optionale Variable “text“ beinhaltet dabei die Bedeutung sowie die zugelassenen Werte für dieses Feld. Die alternativen Parameter include und exclude sind für Klassen und Packages zu verwenden, wenn diese das Interface *Serializable* implementieren.	@serial the date on which the certificate was revoked (gefunden in der Java API, java.security.cert. CertificateRevoked-Exception)
@serialData text	Dieser Tag beschreibt die Sequenzen und die Typen der geschriebenen und gelesen Daten. Kann für writeObject-, readObject-, writeExternal-, readExternal-, writeReplace- und readResolve-Methoden verwendet werden.	@serialData the number of characters currently stored in the string builder ... (gefunden in der Java API, java.lang.StringBuilder)

@serialField name type text	Dokumentiert eine ObjectStreamField-Komponente eines serialPersistentFields-Arrays. Die Variable “name“ gibt dabei den Namen des Feldes, “typ“ den Feldtypen und “text“ die Beschreibung des Feldes an.	@serialField component Component[] The components in this container. (gefunden in der Java API, java.awt.Container)
@since versionsnummer	Dieser Tag gibt an, seit welcher Version dieses Element zur Verfügung steht.	@since 1.5
@throws e text	Mithilfe dieses Tags werden Exceptions und Errors, die durch den Javadocable geworfen werden, beschrieben. Die Variable “e“ beinhaltet dabei den Klassennamen einer solchen Exception / Errors und die Variable “text“ die dazugehörige Beschreibung, wann diese Ausnahme auftreten kann.	@throws IllegalArgumentException wird geworfen, falls der übergebene Wert negativ ist.
@version versionsnummer	Gibt die aktuelle Version des Elements an.	@version 2.0

Tabelle 2.1.: Übersicht über die unterstützten Javadoc-Tags dieser Bachelorarbeit.

2.1.2. Javadocables

Der Begriff Javadocables beschreibt in dieser Bachelorarbeit alle javadocfähigen Elemente. [ORA93b]
Dies umfasst in dieser Arbeit konkret:

- Klassen
- Interfaces
- Enums
- Methoden
- Konstruktoren
- Felder (global)
- Annotations (-typen)
- Pakete (Packages)

Dabei ist zu beachten, dass Pakete ausschließlich in einer extra Datei namens "package-info.java" dokumentiert werden. Diese Datei liegt in dem jeweiligen Root-Verzeichnis des Paketes. Nähere Informationen zu den Javadocables werden am Anfang des dritten Kapitels behandelt.

2.2. Code Smells

Code Smells sind ein Indiz für schlechte Strukturierung in der Programmierung oder im Design. [Kos17]
Deshalb sind festgestellte Code Smells mitunter dafür verantwortlich, dass der Quellcode von Software immer komplizierter und dadurch schwerer zu warten ist. Deshalb sollte an den Stellen im Quellcode, an denen ein Code Smell festgestellt wurde, eine Überarbeitung mittels der Refaktorisierung erfolgen, um spätere Fehler bei einer Modifikation des Codes auszuschließen. [t2i]
Häufig werden Code Smells auch Bad Smells, Software-Erosion oder einfach nur Smells genannt. [Kos17, t2i]
Geprägt wurden diese Smell-Begriffe durch ein Zitat von Kent Beck. Er zitierte seine Oma mit dem Satz "If it stinks, change it", die damals von Babywindeln sprach. [FBB+99, t2i]
Um übelriechenden Code handelt es sich dabei jedoch nur im übertragenen Sinne, denn wenn Martin Fowler, laut seinen eigenen Aussagen, beispielsweise eine zu lange Methode sieht, zuckt ihm die Nase. [Fow06]
Martin Fowler beschrieb in seinem vorhin erwähnten Buch 22 verschiedene Code Smells.
Dabei gibt es verschiedene Arten von Code Smells, die in unterschiedlichen Oberkategorien zusammengefasst werden können. Mäntylä, Vanhanen und Lasssenius beschrieben in ihrem

Artikel zum Thema “A Taxonomy and an Initial Empirical Sutdy of Bad Smells in Code“ sieben solcher Kategorien. Des Weiteren ordneten sie die 22 genannten Code Smells von Fowler in ihren beschriebenen sieben Kategorien ein. [MVL03]

- **Bloaters:** Sie repräsentieren die Code Smells, die zu finden sind, wenn etwas zu groß geworden ist, sodass dieser Code nicht mehr effektiv nutzbar ist. Dieser Kategorie werden “Long Method“, “Large Class“, “Primitive Obsession“, “Long Parameter List“ und “Data Clumps“ zugeordnet.

- **Object-Orientation Abusers:** Wie der Name schon erahnen lässt, sind in dieser Kategorie die Code Smells enthalten, die nicht das volle Potenzial der objektorientierten Programmierung ausnutzen. Ihr wird “Switch Statements“, “Temporary Field“, “Refused Bequest“, “Alternative Classes with Different Interfaces“ und “Parallel Inheritance Hierarchies“ zugeordnet.

- **Change Preventers:** In dieser Kategorie sind Code Smells enthalten, die die Modifikation von Software behindert. Mäntylä et al. ordneten dieser Kategorie “Divergent Change“ und “Shotgun Surgery“ zu.

- **Dispensables:** Die Dispensables sind Code Smells, die etwas Unnötiges im Code repräsentieren, das entfernt werden sollte, wie zum Beispiel duplizierter Code. In dieser Kategorie sind “Lazy Class“, “Data Class“, “Duplicate Code“ und “Speculative Generality“ zu finden.

- **Encapsulators:** In dieser Kategorie sind diejenen Smells enthalten, die sich mit der Datenkommunikation oder der Verkapselung beschäftigen. Von Martin Fowlers Code Smells betrifft dies “Message Chains“ und “Middle Man“.

- **Couplers:** Dieser Kategorie wurden die Code Smells “Feature Envy“ und “Inappropriate Intimacy“ zugewiesen. Diese Kategorie repräsentiert die Code Smells mit einer hohen Kopplungsrate.

- **Others:** Diese Kategorie umfasst alle Code Smells, die nicht eindeutig in eine der anderen Kategorien passt. Ihr wurden die Code Smells “Incomplete Library Class“ und “Comments“ zugeordnet.

Die Code Smells, die von dem Javadoc-Detektor erkannt werden, können allesamt ebenfalls in der Kategorie *“Others“* eingegliedert werden.

2.3. Spoon

Spoon ist eine von INRIA bereitgestellte öffentliche Java-Bibliothek, die 2005/2006 entstand. Diese Bibliothek stellt Quellcode Analysen und Transformationen bereit. Laut INRIA bietet Spoon dabei ein vollständiges und sehr detailliertes Java-Metamodell, in dem alle Programmelemente enthalten sind. Mit einem gegebenen Java-Programm als Eingabe ist es der Spoon-Bibliothek möglich einen dazugehörigen AST in Form dieses Spoon Java-Metamodells zu erzeugen. [PMP+16]
AST steht dabei für Abstract Syntax Tree. Wie ein Teil des Namens schon erahnen lässt ist dieser AST als Baum aufgebaut und bietet somit eine Hierarchie. Abstrakt ist der AST aufgrund dessen, dass nicht jedes Detail der Programmiersprache in dem AST abbildet wird.

Das Hauptinterface des strukturellen Teils des eben genannten Metamodells wird durch das Interface *"CtElement"* repräsentiert. Zusätzlich stellt *CtElement* eine Schnittstelle für Visitoren zur Verfügung, dadurch ist es den Visitoren möglich, die Elemente zu besuchen. Spoon bietet zudem eine Implementation namens *"CtScanner"*, dieser ist mit der Visitoren-Klasse *"CtVisitor"* von Spoon verwandt, dadurch ist es dem Scanner möglich, das Metamodell zu scannen und so die verschiedenen Elemente zu besuchen. [PMP+06]
In der folgenden Abbildung 2.4 [PMP+06] werden die restlichen strukturellen Interfaces des Metamodells gezeigt, dabei sind die Interfaces, die relevant für diese Bachelorarbeit sind, gelb hinterlegt.

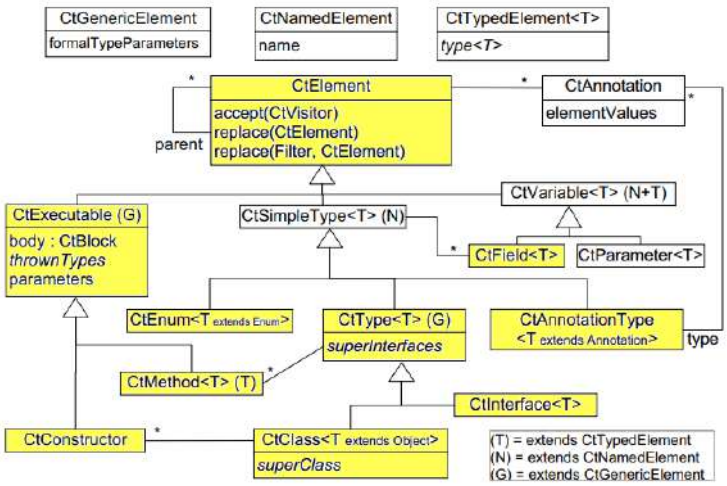


Abbildung 2.4.: Struktureller Teil des Spoon Java Metamodells.

Des Weiteren besteht eine Verbindung zwischen der Elementen- und der Javadoc-Klasse, die Javadoc-Klasse von Spoon beinhaltet dabei den Javadoc-Kommentar des dazugehörigen Elementes.

2.4. LibVCS4j

Der Name LibVCS4j steht für “Libary Version Control System for java“, was soviel bedeutet, wie “eine Bibliothek die mit Versionskontrollsystemen umgehen kann und für Java-Projekte entwickelt wurde“.
LibVCS4j ist, wie in der Motivation schon beschrieben, eine von der Software AG der Universität Bremen zur Verfügung gestellte öffentliche Java-Bibliothek, die Software-Repositorys analysieren kann. Diese Software-Repositorys können unabhängig von ihren Versionskontrollsystemen analysiert werden. Die Bibliothek stellt für verschiedene Versionskontrollsysteme sowie Issue Tracker eine gemeinsame API bereit. Derzeit werden die folgenden Versionskontrollsysteme unterstützt: Git[3], Mercurial[4] und Subversion[5]
Die unterstützten Issue Tracker sind die von Github[6] und von Gitlab[7], jedoch stehen nur bestimmte Authentifizierungsmechanismen zur Verfügung.
Des Weiteren stellt die Bibliothek verschiedene Module zur Analyse dieser Repositorys sowie eine Schnittstelle zur einfachen Entwicklung solcher Module bereit. [Ste18, Ste19]
Der Erkennung von Code Smells dienen einige solcher bereitgestellten Module. Welche Code Smells bereits von LibVCS4j erkannt werden können, kann in der Motivation nachgelesen werden. Die Bibliothek verwendet für ihre Code Smell Detektoren die eben beschriebene Spoon-Bibliothek.
Außerdem integriert die Bibliothek verschiedene Softwares, wie zum Beispiel JGit für den Zugriff auf Repositorys und PMD.
PMD ist ein Tool für die statische Codeanalyse von Quellcodes. Dieses Tool unterstützt viele verschiedene Programmiersprachen, unter anderem auch Java. Durch PMD ist es der Bibliothek möglich weitere Code Smells, die aufgrund von Programmierfehlern entstehen, zu erkennen. Diese Programmierfehler sind zum Beispiel nicht verwendete Variablen, leere Catch-Blöcke oder unnötige Objekterstellungen. [PMD19]

[3] https://git-scm.com/
[4] https://www.mercurial-scm.org/
[5] https://subversion.apache.org/
[6] https://github.com/
[7] https://about.gitlab.com/

3. Implementierung

Aufgrund der Wichtigkeit eines sinnvollen und fehlerfreien Dokumentationskommentars, die im vorherigen Kapitel vermittelt wurde, ist es umso offensichtlicher, dass ein Detektor benötigt wird, der die vorhanden Javadoc-Kommentare analysiert und auf dessen Fehlern aufmerksam macht. Und genau das bietet der Javadoc-Detektor, der dieser Arbeit zugrunde liegt. Wie dieser Detektor konkret funktioniert, wird in diesem Kapitel festgehalten.
Als Erstes werden kleine Definitionen für den Detektor festgelegt und anschließend die Konfigurationsmöglichkeiten des Detektors beschrieben. Darauf findet im Unterkapitel über die "zulässigen Javadoc-Tags für die jeweiligen Javadocables" eine Zuordnung dieser Tags zu den Javadocables statt. Danach erfolgt die Funktionsbeschreibung des Javadoc-Detektors. Anschließend wird erläutert wie und welche Javadoc Code Smells erkannt werden können.

Der Begriff "Typen" bezeichnet unter anderem zusammenfassend die Javadocables Klassen sowie Interfaces und Enums.[Ull12, S. 1259]
Annotationen sind eigentlich ebenfalls "Typen" [LL14], werden aber von dem Detektor aufgrund seiner zu unterschiedlichen zulässigen Javadoc-Tags gesondert behandelt. Aufgrund dessen umfasst der Begriff "Typen" in dieser Arbeit nur die ersten drei genannten Typen (Klassen, Interfaces und Enums).
Mit dem Begriff "Top-Level-Typen" [GJS+15, Kap. 7.6] sind Klassen und Interfaces der obersten Ebene gemeint, sie sind folglich nicht in einem anderen Typen geschachtelt.
Des Weiteren sind mit Annotations die spezielleren Annotationstypen [GJS+15, Kap. 9.6] gemeint. Diese legen Annotationen fest und oberhalb eines solchen Typen wird die Dokumentation der jeweiligen Annotation festgehalten.
Als Executables werden zusammenfassend Methoden und Konstruktoren bezeichnet.[1]
Felder, oder auch Variablen genannt, können in zwei Kategorien unterteilt werden, zum einen die lokalen und zum anderen die globalen Variablen. Dabei sind jedoch nur die globalen Felder für eine Dokumentation relevant, wie ebenfalls aus der Abbildung 2.1 und Abbildung 2.2 entnommen werden kann. Globale Variablen beschreiben dabei die Klassen- und die Instanz-Variablen. [Wag12]
Der Begriff "Field-Access" umfasst zusammenfassend die Getter- und Setter-Methoden.
Außerdem sind mit typenlose Pakete Pakete gemeint, die keine Klasse, Interface, Enum oder ein Annotationstypen beinhalten beziehungsweise deklarieren. Diese Pakete deklarieren lediglich weitere Unterpakete. Ein Beispiel hierzu ist das Paket *java*, dieses Paket deklariert nur weitere Unterpakete. Einige dieser Unterpakete sind unter anderem das Paket *util* oder das Paket *lang*.

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Executable.html

3.1. Konfigurationsmöglichkeiten

Dieses Unterkapitel widmet sich den Konfigurationsmöglichkeiten, die der Javadoc-Detektor bietet.
Aufgrund dessen, dass es keine einheitliche Vorgabe gibt, ab wie vielen Zeichen oder Wörtern ein Javadoc-Kommentar als hinreichend beschrieben gilt, ist es notwendig, dass der Javadoc-Detektor nach den jeweiligen Kriterien des Benutzers konfiguriert werden kann.
Um eine Option für den Detektor zu konfigurieren, wird die jeweilige Option mit einem *"with"* eingeleitet, wenn eine Option entfernt werden soll, wird sie mit einem *"without"* eingeleitet. Zum Beispiel konfiguriert *"withParam(5)"* die Option, dass Parameter mit fünf Wörtern beziehungsweise Charakter, je nachdem welche Zählvariante gewählt wurde, beschrieben werden muss.
Direkt nach der Initialisierung des Detektors sind nur sehr wenige Konfigurationen voreingestellt. Zu diesen Konfigurationen gehören das wortweise Zählen sowie die Konfiguration, dass Getter und Setter dokumentiert werden müssen und das Pakete inklusive die typenlosen Pakete keine Package-info-Datei benötigen. Die restlichen Konfigurationen sind nicht konfiguriert. Eine nicht konfigurierte Mindestlänge besitzt die Standardmindestlänge von 0. Aufgrund dessen, müssen zwingend mindestens ein Javadocable, ein Zugriffsmodifikator und die jeweiligen Mindestlängen für die Beschreibung und der Tags, die untersucht werden sollen, konfiguriert werden, damit der Detektor funktionsfähig ist. Dabei bietet der Detektor für folgende Optionen Konfigurationsmöglichkeiten.

- **Javadocables:** Mithilfe dieser Konfigurationen werden die Javadocables, die von dem Detektor analysiert werden sollen, konfiguriert. Im Detektor wird mittels einer Sammlung festgehalten, welche Javadocables untersucht werden sollen. Dabei werden dem Benutzer drei verschiedene Möglichkeiten zum Hinzufügen und zum Entfernen der Javadocables zur Verfügung gestellt. Zum einen kann mittels der Methoden *withAllJavadocables()* und *withoutAllJavadocables* alle verfügbaren Javadocables auf einmal hinzugefügt oder entfernt werden. Zum anderen kann mittels
  *withJavadocable(**einzelnerJavadocable**)* oder *withoutJavadocable(**einzelnerJavadocable**)* gezielt ein Javadocable hinzugefügt oder entfernt werden. Außerdem kann durch die Methoden *withJavadocable(**liste von javadocables**)* oder
  *withoutJavadocable(**liste von javadocables**)* mehrere Javadocables hinzugefügt oder entfernt werden.

- **Zugriffsmodifikatoren:** Durch diese Konfigurationsoptionen werden die Zugriffsmodifikatoren der Javadocables festgelegt, die bei der Analyse berücksichtigt werden sollen. Wie auch bei der eben beschriebenen Konfigurationsoptionen für die Javadocables, werden auch die Zugriffsmodifikation in einer Sammlung gespeichert. Des Weiteren bieten diese Konfigurationsoptionen für die Zugriffsmodifikatoren ebenfalls drei Möglichkeiten, um diese zu konfigurieren. Diese Möglichkeiten funktionieren analog zu den eben beschriebenen. Die Methoden dafür lauten: *withAllAccessModifier()* und *withoutAllAccessModifier()* sowie *withAccessModifier(**einzelnerAccessModifier**)*
  und *withoutAccessModifier(**einzelnerAccessModifier**)*, oder

*withAccessModifier(**liste von accessModifier**)* und
*withoutAccessModifier(**liste von accessModifier**)*.
Dabei werden zwischen den Zugriffsmodifikatoren *public*, *private*, *protected* und *null* unterschieden. *Null* repräsentiert dabei den Default beziehungsweise den package-private Zugriffsmodifikator.

- **Mindestlängen der jeweiligen Javadoc-Tags:** Mittels diesen Konfigurationsoptionen werden die Mindestlängen der Javadoc-Tags konfiguriert oder entfernt. Dafür werden für jeden Tag jeweils zwei verschiedene Methoden bereitgestellt. Zum Konfigurieren der Mindestlänge wird *withTagname(**mindestlänge**)* verwendet und zum Entfernen der Mindestlänge *withoutTagname()*. Dabei wird der Tagname durch den jeweiligen Tag-Bezeichner, ohne dem @-Zeichen, ersetzt. Ein Beispiel hierfür wäre *withDeprecated(5)*. Diese Konfiguration besagt, dass der Javadoc-Tag *@deprecated* mit fünf Zeichen oder Wörtern, je nachdem welche Zählweise gewählt wurde, beschrieben werden muss. Mit *withoutDeprecated()* wird die konfigurierte Mindestlänge wieder auf die Standardmindestlänge 0 gesetzt. Bei den Javadoc-Tags *@param*, *@throws* und *@exception* werden die zusätzlichen Parameter, wie der Exception- oder Error-Klassenname und der Variablenbezeichner nicht zu der Mindestlänge mit gezählt.
  Die zur Verfügung stehenden Tags wurden im zweiten Kapitel im Abschnitt Javadoc-Tags beschrieben.

- **Mindestlängen der Beschreibungen:** Für diesen Konfigurationsbereich werden für jeden Javadocable jeweils neun verschiedene Methoden bereitgestellt. Diese neun Methoden verhalten sich für jeden Javadocable analog, weshalb diese im Folgenden nur für einen Javadocable erläutert werden. Lediglich die Pakete aktivieren in ihren *withPackage(..)* - Methoden noch zusätzlich die Konfigurationsmöglichkeit *"Paket benötigt Package-info"*. Diese zusätzliche Konfigurationsmöglichkeit wird im Folgenden genauer beschrieben. In der dazugehörigen *withoutPackage()* - Methode wird diese zusätzliche Konfigurationsmöglichkeit mit deaktiviert.
  Im Folgenden werden die bereitgestellten Methoden für die Javadocables erläutert, dabei wird die Bezeichnung "**JavadocableName**" durch den jeweiligen Javadocable-Bezeichner ersetzt, den er konfiguriert.
  Die zwei Methoden *with**JavadocableName(minShort, minLong)*** und
  *with**JavadocableName(minTotal)*** unterscheiden sich lediglich dadurch, dass die erstgenannte Methode die Mindestlänge der Kurz- und Langbeschreibung konfiguriert und die zweitgenannte die der Gesamtbeschreibung. Darüber hinaus wird der jeweilige Javadocable zu der Sammlung der zu analysierenden Javadocables, wie eben in der Javadocables-Konfigurationsmöglichkeit beschrieben, hinzugefügt.
  Mittels *without**JavadocableName()*** wird die Längenkonfiguration für den jeweiligen Javadocable zurückgesetzt. Zusätzlich wird der Javadocable aus der Sammlung der zu untersuchenden Javadocables entfernt. Die verbleibenden sechs Methoden sind zum reinen Konfigurieren der jeweiligen einzelnen Mindestlängen gedacht:
  *with**JavadocableName**Short(**minShort**)*, *without**JavadocableName**Short()*,
  *with**JavadocableName**Long(**minLong**)*, *without**JavadocableName**Long()*,

*with**JavadocableName**Total(**minTotal**)* und *without**JavadocableName**Total()*

Des Weiteren werden für den Javadocable Methoden drei weitere Methoden zur Konfiguration der Längen von Field-Access bereitgestellt.
Dabei setzt *withMethodFieldAccess(**minShort, minLong**)*, sowohl die Mindestlänge der Kurzbeschreibung auf den Wert *"minShort"* sowie die Mindestlänge für die Langbeschreibung auf den Wert *"minLong"*. Außerdem werden Methoden zu den zu analysierenden Javadocables hinzugefügt und die Konfiguration, dass ein Field-Access dokumentiert werden muss aktiviert. Die Methode *withMethodFieldAccess(**minTotal**)* funktioniert dabei analog zu der eben genannten Methode, mit dem Unterschied, dass statt der Kurz- und Langbeschreibung die Gesamtbeschreibung eine Mindestlänge zugewiesen bekommt.
Mittels *withoutMethodFieldAccess()* werden diese Einstellungen zurückgesetzt, jedoch werden Methoden nur von den zu untersuchenden Javadocables entfernt, sofern Methoden keine eigene konfigurierte Mindestlänge besitzen.
Des Weiteren bietet der Detektor auch einige Möglichkeiten, um mehrere Javadocables auf einmal zu konfigurieren. Dabei bietet jede dieser Möglichkeiten ebenfalls die neun genannten verschiedenen Methoden. Eine Möglichkeit um die Länge der Beschreibung aller verfügbaren Javadocables zu konfigurieren, ist mittels des Javadocable-Bezeichners "All". Des Weiteren gibt es die Möglichkeiten die Länge der Beschreibung für die Typen (Klassen, Interface, Enum) mittels des Bezeichners "Type" und die Executables mittels des Bezeichners "Executable" zu konfigurieren. Die Methoden dieser drei zusammenfassenden Möglichkeiten funktionieren dabei analog zu denen, die nur die Länge eines einzelnen Javadocables konfigurieren.

- **Zählen via Wörter / Charakter:** Standardmäßig ist für diese Konfiguration das wortweise Zählen aktiviert. Dies kann ebenfalls durch die Methode *withReadByWord()* erzeugt werden. Das wortweise Zählen zählt neben den Wörtern, die in dem Text vorkommen, auch Zahlen als ein Wort. Die zuzählenden Wörter werden dabei durch ein oder mehreren Leerzeichen oder Bindestriche getrennt. Alternativ lässt sich mittels der Methode *withoutReadByWord()* das charakterweises Zählen aktivieren. Bei dem charakterweises Zählen werden die Leerzeichen in der Beschreibung ebenfalls mitgezählt.

- **Field-Access:** Diese Konfiguration ist standardmäßig so eingestellt, dass Getter- und Setter-Methoden dokumentiert werden müssen. Um zu erkennen, ob eine Methode ein Field-Access ist, wird die bereitgestellte Methode *isFieldAccess(**Methode**)* von der Klasse Scanner verwendet. Mittels *withFieldAccessMustBeDocumented()* kann die Option, dass ein Field-Access dokumentiert sein muss, aktiviert und mit der Methode *withoutFieldAccessMustBeDocumented()* deaktiviert werden.

- **Paket benötigt Package-info:** Da die Dokumentation eines Paketes in seiner jeweiligen Package-info-Datei zu finden ist und ein Paket ohne Package-info nicht analyisert werden kann, wird diese Konfiguration benötigt. Diese Konfiguration ermöglicht dem Detektor, dass er die Pakete, die von einem Typen oder Annotation verwendet

werden, hinsichtlich der Existenz ihrer Package-info-Datei überprüfen kann. Mittels *withPackageInfoNeeded()* kann die Konfiguration aktiviert und mittels *withoutPackageInfoNeeded()* deaktiviert werden. Des Weiteren wird diese Konfiguration ebenfalls aktiviert, sobald der Detektor den Javadocable *"Pakete"* analysieren soll.

- **Typenloses Paket benötigt Package-info:** Diese Konfiguration ist eine Verfeinerung zu der eben genannten Konfigurationsmöglichkeit. Durch diese Konfiguration ist es dem Detektor zusätzlich möglich auch die Pakete hinsichtlich der Existenz ihrer Package-info-Datei zu überprüfen, die keinen Typen und keine Annotation deklarieren. Mittels der vorherigen Konfiguration werden nur die Pakete auf ihre Package-info untersucht, die über eine Analyse ihrer deklarierten Typen oder Annotations gefunden werden. Ein typenloses Pakete kann über die Deklaration eines Unterpaketes gefunden werden. Ein Beispiel hierfür wäre, wenn der Detektor die Klasse *"Boolean"* aus dem Paket *"java.lang"* untersucht, dann ist das Paket *"java"* das deklarierende Paket des Paketes *"java.lang"*. Dadurch würde das Paket *"java"* hinsichtlich der Existenz seiner Package-info-Datei untersucht werden. Diese Untersuchung der typenlosen Pakete wird so lange fortgeführt, bis der Detektor das oberste Paket der Deklarationen überprüft hat. Aktiviert kann diese Konfiguration mittels *withTypelessPackageInfoNeeded()* werden und deaktiviert mittels *withoutTypelessPackageInfoNeeded()*.

Konfigurationsbeispiel:

```
1  javadocDetector.withAll(15);
2  javadocDetector.withEnum(10);
3  javadocDetector.withField(5);
4  javadocDetector.withExecutable(10);
5  javadocDetector.withMethodFieldAccess(5);
6  javadocDetector.withAllAccessModifier();
7  javadocDetector.withPackageInfoNeeded();
8  javadocDetector.withTypelessPackageInfoNeeded();
9  javadocDetector.withDeprecated(5);
10 javadocDetector.withReturn(3);
11 javadocDetector.withParam(3);
12 javadocDetector.withThrows(5);
13 javadocDetector.withAuthor(2);
14 javadocDetector.withVersion(1);
15 javadocDetector.withSee(1);
16 javadocDetector.withSince(1);
17 javadocDetector.withSerial(5);
18 javadocDetector.withSerialField(5);
19 javadocDetector.withSerialData(5);
```

Listing 3.1: Ein Beispiel für eine Konfiguration des Javadoc-Detektors.

Das Listing 3.1 zeigt eine Konfiguration eines Javadoc-Detektors, diese wird auch für die Evaluation der Forschungsfragen verwendet. Dabei werden für alle Javadocables die Mindestlänge für die Gesamtbeschreibung (Total-Description) konfiguriert. Als Erstes wird die Mindestlänge für alle Javadocables auf fünfzehn Wörtern festgelegt und anschließend werden

für einige Javadocables eine speziellere Mindestlänge konfiguriert. Außerdem wird konfiguriert, dass alle Zugriffsmodifikatoren berücksichtigt werden sollen. Des Weiteren werden die Konfigurationsmöglichkeiten *"Paket benötigt Package-info"* und *"Typenlose Pakete benötigen Package-info"* aktiviert. Danach folgen die Konfigurationen für die einzelnen Mindestlängen der Javadoc-Tags. Zusätzlich wurden die Einstellungen *"wortweises Zählen"* und *"Field-Access benötigt Dokumentation"*, die bereits zur Initialisierung aktiviert sind, nicht geändert. Des Weiteren werden die jeweiligen Mindestlängen für die einzelnen Javadoc-Tags mit den im Beispiel beschrieben Werten konfiguriert.

3.2. Zugelassene Javadoc-Tags für die jeweiligen Javadocables

Dieses Kapitel behandelt das Thema, welche Javadoc-Tags für welchen Javadocable zulässig sind. Dieses Thema wird mittels der folgenden Tabelle beantwortet. Die Informationen hierfür wurden von der Javadoc-Dokumentation aus dem Absatz "Where Tags Can Be Used" von Oracle [ORA93b] entnommen und für diesen Detektor angepasst.

Javadocables / Javadoc-Tags	Annotation	Class	Interface	Enum	Method	Constructor	Field	Package
@author	x	x	x	x				x
@deprecated		x	x	x	x	x	x	
@exception					x	x		
@param		x	x		x	x		
@return					x			
@see	x	x	x	x	x	x	x	x
@serial		x	x	x			x	x
@serialData					x			
@serialField							x	
@since	x	x	x	x	x	x	x	x
@throws					x	x		
@version	x	x	x	x				x

Tabelle 3.1.: Übersicht der zulässigen Javadoc-Tags für die jeweiligen Javadocables

Der Javadoc-Tag *"@param"* wird für Klassen und Interfaces nur aufgrund der generischen Parameter gestattet. Außerdem ist der Javadoc-Tag @serial für Klassen und Packages nur in Verbindung mit dem Parameter *include* oder *exclude* möglich.

3.3. Javadoc Code Smell - Arten

In diesem Abschnitt geht es um die Frage, welche Javadoc Code Smells von dem Javadoc-Detektor erkannt werden können.
Der Javadoc-Detektor ist in der Lage 37 verschiedene Javadoc Code Smells zu erkennen. Diese 37 Smells können dabei in verschiedene Oberkategorien eingeordnet werden.
Diese Oberkategorien umfassen dabei:

- **No Javadoc:** Diese Kategorie enthält alle Javadoc Code Smells, die gefunden werden, wenn das jeweilige Element keinen Javadoc-Kommentar enthält.
- **Missing tag:** In dieser Kategorie sind die Smells zu finden, die gefunden werden, wenn ein Javadoc-Tag in dem Javadoc-Kommentar fehlt.
- **Missing tag requirement:** Diese Kategorie umfasst die Code Smells, die gefunden werden, wenn die jeweilige Tag-Voraussetzung nicht gegeben ist.
- **Too short description:** Enthält alle Code Smells, die mit der Beschreibung des Javadoc-Kommentars zusammenhängen.
- **Too short tag description:** Enthält alle Code Smells, die durch zu kurze Tag-Beschreibung erkannt werden.
- **Missing package-info:** Diese Kategorie enthält die beiden Smells, die zu der Package-info Konfiguration gehören.
- **Tag is not allowed:** In dieser Kategorie ist lediglich der Code Smell zu finden, der besagt, dass der analysierte Javadoc-Tag nicht für den jeweiligen Javadocable zulässig ist.

In der folgenden Tabelle werden diese Code Smells näher beschrieben. Außerdem werden in der Tabelle die verschiedenen Oberkategorien farblich voneinander getrennt.

Javadoc Code Smells	**Bescheibung**
AnnotationType *annotationTypeName* has no javadoc.	Der besuchte Annotationstyp besitzt keinen Javadoc-Kommentar. Dabei beschreibt *annotationTypeName* den Annotationnamen.
Class *className* has no javadoc.	Die besuchte Klasse besitzt keinen Javadoc-Kommentar. Dabei gibt *className* den Klassennamen an.
Constructor *constructorSignature* has no javadoc.	Der besuchte Konstruktor besitzt keinen Javadoc-Kommentar. Dabei gibt *constructorSignature* den Namen des Konstruktors sowie die dazugehörigen Parameter an.
Enum *enumName* has no javadoc.	Der besuchte Enum besitzt keinen Javadoc-Kommentar. Dabei gibt *enumName* den Namen des Enums an.
Field *fieldName* has no javadoc.	Das besuchte Feld besitzt keinen Javadoc-Kommentar. Dabei gibt *fieldName* den Namen dieses Feldes an.
Interface *interfaceName* has no javadoc.	Das besuchte Interface besitzt keinen Javadoc-Kommentar. Dabei gibt *interfaceName* den Namen dieses Interfaces an.
Method *methodSignature* has no javadoc.	Die untersuchte Methode besitzt keinen Javadoc-Kommentar. Dabei gibt *methodSignature* den Methodennamen sowie die dazugehörigen Parameter an.
Package *packageName* has no javadoc.	Das besuchte Paket besitzt keinen Javadoc-Kommentar. Dabei gibt *packageName* den Namen dieses Paketes an. Dies funktioniert jedoch nur, wenn eine Package-info für das jeweilige Paket existiert.
Missing tag @author in the javadoc.	Dieser Code Smell tritt ein, wenn die erwartete Autoren-Tag-Länge gesetzt wurde, es sich bei dem besuchten Javadocable um einen Top-Level-Typen handelt und der Autoren-Tag nicht beschrieben wurde.
Missing tag @version in the javadoc.	Dieser Code Smell tritt ein, wenn die erwartete Version-Tag-Länge gesetzt wurde, es sich bei dem besuchten Javadocable um einen Top-Level-Typen handelt und der Versions-Tag nicht beschrieben wurde.

Missing tag @deprecated in the javadoc.	Dieser Code Smell tritt ein, wenn der Javadocable von dem untersuchten Javadoc die Annotation @Deprecated an dessen Deklaration besitzt, jedoch nicht den Javadoc-Tag @deprecated im Javadoc beschreibt.
Missing tag @return in the javadoc.	Dies ist der Fall, wenn die Methode einen Rückgabetyp besitzt, jedoch diesen in den Javadoc nicht dokumentiert.
Missing tag @param *parameterName* in the javadoc.	Tritt ein, wenn ein Parameter *parameterName* nicht beschrieben wurde. Neben den Parametern aus der Parameterliste des jeweiligen Executables funktioniert dieser Smell auch für generische Parameter. Klassen und Interfaces können ebenfalls generische Parameter beschreiben.
Missing tag @throws *exceptionName* in the javadoc.	Dies ist der Fall, wenn eine Exception oder ein Error, die von dem jeweiligen Executable geworfen werden kann, nicht beschrieben wurde. Dabei werden auch die Exceptions und Errors, die durch aufgerufene Executables rekursiv geworfen werden können berücksichtigt.
Javadoc contains @deprecated, but the annotation @Deprecated at the javadocable is missing.	Dieser Code Smell tritt ein, wenn der Javadoc-Kommentar den Tag @deprecated beschreibt, jedoch der Javadocable an seiner Deklaration nicht die Annotation @Deprecated besitzt.
Javadoc contains @return, but the method has no return value.	Dies ist der Fall, wenn der Javadoc den Tag @return beschreibt, jedoch die Methode keinen Rückgabetypen deklariert hat und somit eine *"void"* Methode ist.
Javadoc contains @param parameterName, but this parameter does not exists.	Der Javadoc-Kommentar enthält die Dokumentation eines Parameter mit dem Namen *parameterName*, jedoch existiert dieser Parameter nicht im dazugehörigen Javadocable. Dieser Smell funktioniert ebenfalls für generische Parameter.
Javadoc contains @throws/@exception exceptionName, but this does not thrown by this or a called executable.	In diesem Fall beschreibt der Javadoc-Kommentar eine Exception oder ein Error mittels des Tags @throws oder @exception, die nicht von dem Executable geworfen werden kann.
No description existing in this javadoc.	Dieser Code Smell tritt ein, wenn der Javadocable einen keine Javadoc-Beschreibung enthält. Dies kann nur erkannt werden, sofern eine Mindestlänge für die Short-, Long- oder Total-Description gesetzt wurde.

Short-description of this javadoc is too short.	Dieser Smell wird erkannt, sobald die gefundene Short-Description kürzer ist als die konfigurierte Mindestlänge für Short-Descriptions.
Long-description of this javadoc is too short.	Dieser Smell wird erkannt, sobald die gefundene Long-Description kürzer ist als die konfigurierte Mindestlänge für Long-Descriptions.
Total-description of this javadoc is too short.	Dieser Smell wird erkannt, sobald die gefundene Total-Description kürzer ist als die konfigurierte Mindestlänge für Total-Descriptions.
Description of the @author tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @author-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @deprecated tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @deprecated-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @exception exceptionName is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @exception *exceptionName* - Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @param parameterName is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @param *parameterName* - Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @return tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @return-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @see tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @see-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @serial tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @serial-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @serialData tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @serialData-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @serialField tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @serialField-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @since tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @since-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.

Description of the @throws exceptionName is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @throws *exceptionName* - Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Description of the @version tag is too short.	Die Beschreibung des in dem Javadoc-Kommentar gefundenen @version-Tag ist kürzer als die konfigurierte Mindestlänge für diesen Tag.
Missing package-info for the package*qualifiedName*.	Bei dem Paket mit dem vollqualifizierten Namen *qualifiedName* fehlt die package-info-Datei und somit deren Dokumentation.
Missing package-info for the typeless package *qualifiedName*.	Bei dem Paket mit dem vollqualifizierten Namen *qualifiedName* fehlt die package-info-Datei und somit deren Dokumentation. Typeless bedeutet hierbei, dass dieses Paket keine Typen oder Annotationen beinhaltet.
Tag *tagType* is not allowed for this javadocable.	Der beschriebene Javadoc-Tag vom Typ des *tagType* ist für diese Art des Javadocables nicht zulässig.

Tabelle 3.2.: Die erkennbaren Javadoc Code Smells des Javadoc-Detektors.

Dass die Annotation @Deprecated und der Javadoc-Tag @deprecated zusammen auftreten, kann in der Oracle “Deprecated“-Dokumentation im Abschnitt “How to Deprecate“ entnommen werden. [ORA93a]
Durch die Annotation @Deprecated werden die Entwickler in der IDE darauf hingewiesen, dass dieses Element veraltet ist. Und mittels des Javadoc-Tag können dem Entwickler Hinweise darüber gegeben werden, welche Funktionalität er stattdessen verwenden soll.

3.4. Funktionsweise des Javadoc-Detektors

Dieses Unterkapitel beschäftigt sich mit dem Aufbau sowie der Funktionsweise des Javadoc-Detektors.
Der Javadoc-Detektor erbt von der Klasse “CodeSmellDetector“. Die Klasse “CodeSmellDetector“ wird von der LibVCS4j-Bibliothek zur Verfügung gestellt, um neue Detektoren zu implementieren. Dieser erbt wiederum von der Klasse “Scanner“. Die Klasse “Scanner erweitert die im zweiten Kapitel erwähnte Spoon-Klasse “CtScanner“. Durch diese Erbschaft ist es den Detektoren möglich, den von Spoon generierten und bereitgestellten AST zu scannen. Des Weiteren stellt der CodeSmellDetector eine Sammlung der von dem Detektor erkannten Code Smells zur Verfügung. Für die Code Smells existiert eine von der LibVCS4j-Bibliothek bereitgestellte gleichnamige Klasse. Jede Instanz dieser Klasse enthält unter anderem die Informationen über die Position des Javadocables, in dessen Javadoc-Kommentar der Code Smell gefunden wurde. Des Weiteren enthält diese “Code Smell“-Instanz die Signatur des Javadocables und eine zusammenfassende Beschreibung des jeweiligen Smells. Dabei ist mit der Position gemeint, an welcher Stelle sich der jeweilige Javadocable im Quellcode befindet. Die jeweiligen Instanzen der Klasse “CodeSmell“ bieten darüber hinaus noch weitere Informationen, die für den Javadoc-Detektor nicht relevant sind.
Bevor der Javadoc-Detektor seine Analyse starten kann, muss er, wie im Kapitel Konfigurationsmöglichkeiten beschrieben, konfiguriert werden.
Um die Funktionsweise in diesem Kapitel besser nachvollziehen zu können, wird angenommen, dass die Konfiguration so vorgenommen wurde, dass jeder Javadocable mit jedem Zugriffsmodifikator untersucht werden soll sowie dass sämtliche Längen für die Beschreibungen und Javadoc-Tags konfiguriert wurden. Des Weiteren wird angenommen, dass Getter und Setter dokumentiert sein müssen sowie dass jedes Paket eine Package-info-Datei benötigt.

Nachdem die Analyse des Javadoc-Detektors gestartet wurde, besucht dieser zunächst die verschiedenen Javadocables. Die Überprüfgen, die der Detektor dabei vornimmt, sind von der Kernfunktionalität bei allen acht Javadocables identisch. In dem folgenden Aktivitätsdiagramm 3.1 wird ein solcher Ablauf demonstriert.

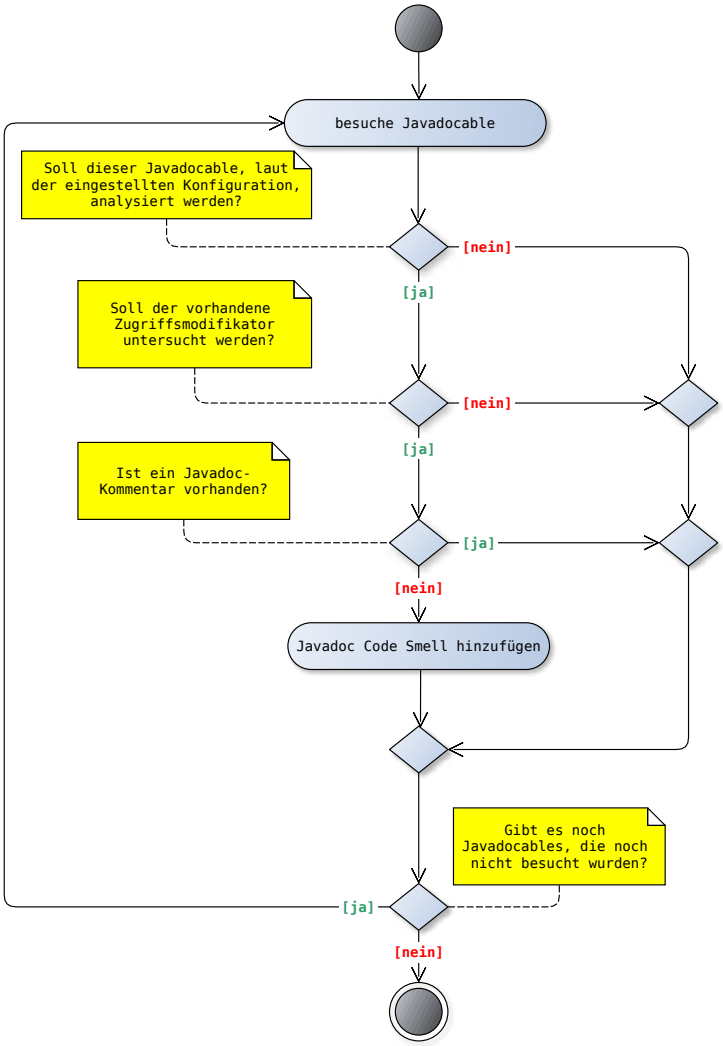


Abbildung 3.1.: Aktivitätsdiagramm - genereller Ablauf der Erkennung des Code Smells *"no javadoc"*.

Als Erstes wird überprüft, ob der vorhandene Javadocable laut den vorgenommenen Konfigurationen untersucht werden soll. Wenn dies der Fall ist, wird untersucht, ob der vorhandene Zugriffsmodifikator ebenfalls in der Konfiguration als zu untersuchender Zugriffsmodifikator konfiguriert wurde. Sofern beide dieser Bedingungen zutreffen, handelt es sich generell um einen Javadocable der dokumentiert werden soll und somit einen Javadoc-Kommentar besitzen muss.
Als Nächstes wird überprüft, ob der Javadocable einen Javadoc-Kommentar besitzt. Falls dies nicht der Fall ist, wird ein *"no Javadoc"* - Code Smell des jeweiligen Javadocables von dem Detektor erkannt. Anschließend wird der nächste Javadocable besucht, solange bis alle Javadocables des AST besucht wurden. Für Felder und Konstruktoren genügt dieser generelle Ablauf der *"no Javadoc"* - Erkennung aus, für die restlichen Javadocables sind noch zusätzliche Bedingungen relevant.

Bei den Methoden gibt es zu den ersten beiden Bedingungen noch zusätzliche Bedingungen, die beachtet werden müssen, um Aussagen darüber treffen zu können, ob jene Methode dokumentiert sein muss oder nicht. Dabei muss nach den ersten beiden Bedingungen geprüft werden, ob es sich bei dieser Methode um einen Field-Access, also um eine Getter- oder Setter-Methode, handelt. Wenn dies der Fall ist, muss überprüft werden, ob die Konfiguration "Field-Access benötigt eine Dokumentation" aktiviert ist.
Handelt es sich bei der Methode nicht um eine Field-Access-Methode, dann benötigt die Methode immer einen Javadoc-Kommentar. Wenn die Methode allerdings eine Field-Access-Methode ist und zusätzlich wurde die eben genannte Konfiguration eingestellt, dann benötigt auch diese Field-Access-Methode einen Javadoc-Kommentar. Wie in dem generellen Ablauf folgt nun die Überprüfung, ob ein Javadoc-Kommentar vorhanden ist und es wird gegebenenfalls ein *"no Javadoc"* Code Smell von dem Detektor erkannt. Danach erfolgt die Untersuchung des nächsten Javadocables oder das Ende dieser Überprüfung. Das Aktivitätsdiagramm 3.2 ist das passende Aktivitätsdiagramm hierzu.

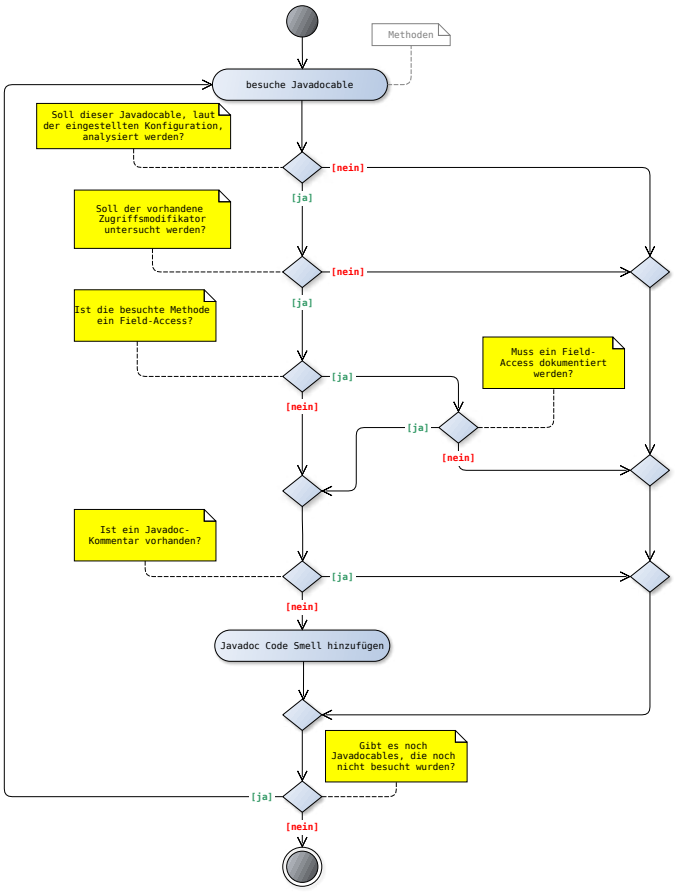


Abbildung 3.2.: Aktivitätsdiagramm - Ablauf für Methoden für die Erkennung des *"no Javadoc"* - Code Smells.

Typen und Annotations befolgen den oben genannten generellen Ablauf der *“no Javadoc“* - Erkennung. Mit dem Unterschied, dass noch zusätzliche Überprüfungen für die Konfigurationen *“Paket benötigt Package-info“* und *“Typenloses Paket benötigt Package-info“* erfolgen, bevor der nächste Javadocable untersucht wird. Im folgenden Aktivitätsdiagramm 3.3 kann dieser Programmablauf veranschaulicht werden.

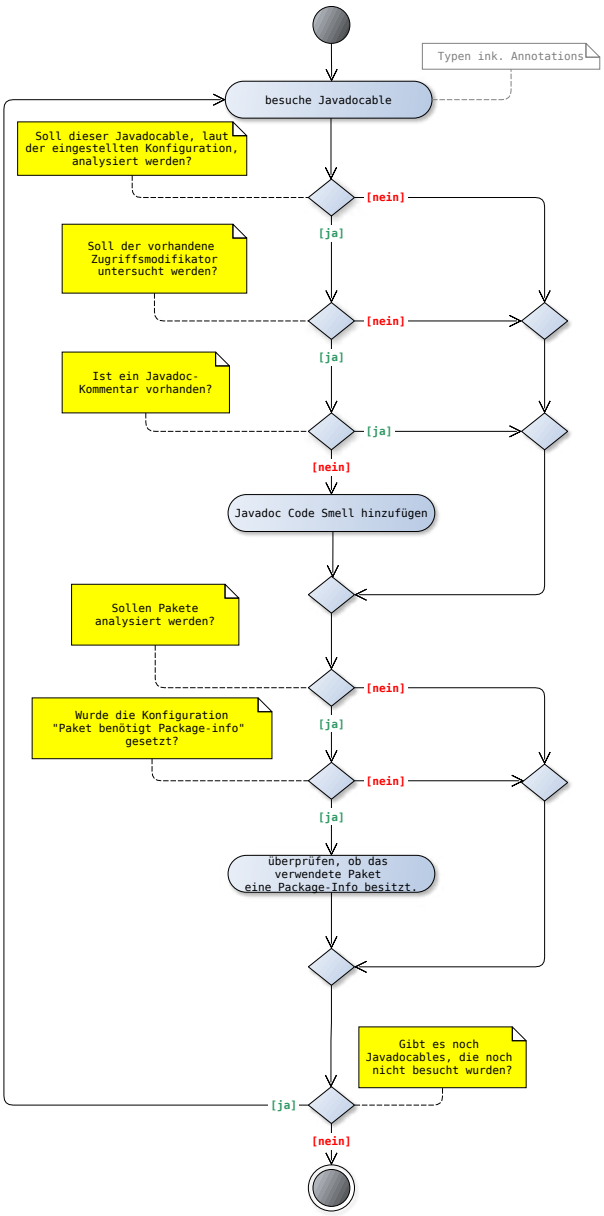


Abbildung 3.3.: Aktivitätsdiagramm - Ablauf für Typen und Annotationstypen für die Erkennung des *“no Javadoc“* - Code Smells.

Dabei wird zuerst überprüft, ob der Javadocable Pakete analysiert werden soll und anschließend, ob die eben genannte Konfiguration *"Paket benötigt Package-info"* eingestellt wurde. Wenn beides der Fall ist, wird überprüft, ob das verwendete Paket eine Package-info-Datei besitzt. Ist dies nicht der Fall, wird ein entsprechender Code Smell von dem Detektor erkannt. Des Weiteren wird die Konfiguration *"Typenloses Paket benötigt Package-info"* ebenfalls in der Überprüfung überprüft. Durch diese erkennt der Detektor den Code Smell *"Missing package-info for the typeless package"* für typenlose Pakete ohne Package-info-Datei. Somit werden auch typenlose Pakete ohne Dokumentation erkannt.

Die Erkennung der *"no Javadoc"* - Smells von Paketen befolgt den im folgenden Aktivitätendiagramm 3.4 gezeigten Ablauf.

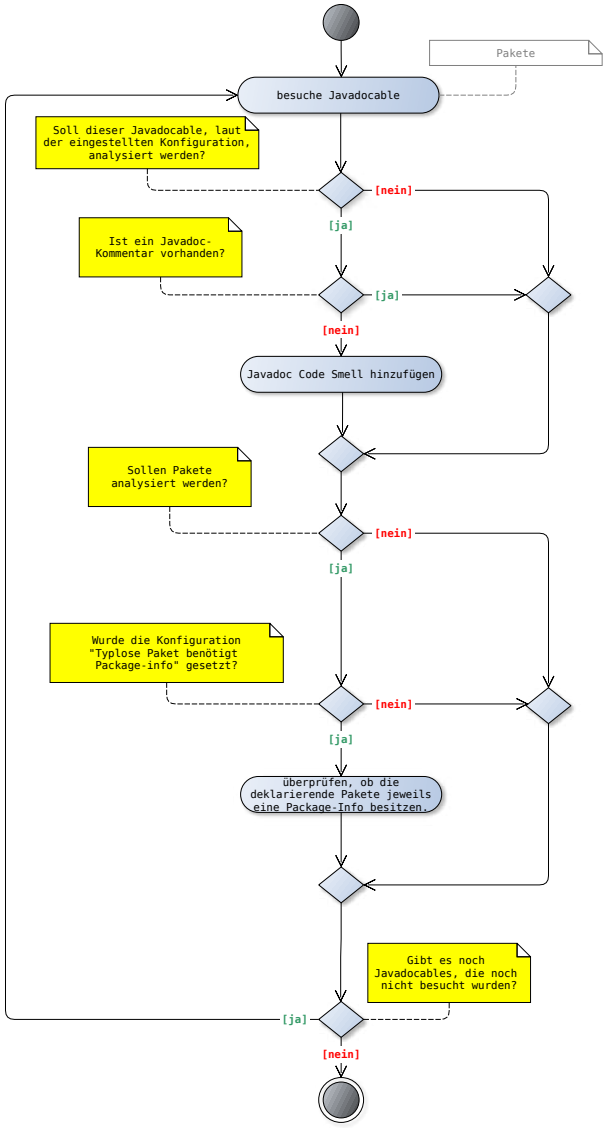


Abbildung 3.4.: Aktivitätsdiagramm - Ablauf für Pakete für die Erkennung des *"no Javadoc"* - Code Smells.

Bei dieser Untersuchung entfällt die Überprüfung des Zugriffsmodifikators, da Pakete keinen Zugriffsmodifikator besitzen. Ansonsten befolgt es dieselben Überprüfungen wie bei dem oben genannten generellen Ablauf. Zusätzlich werden, sofern die Konfiguration *"Typenlose Pakete benötigen Package-info"* aktiviert wurde, typenlose Pakete hinsichtlich ihrer Package-info-Dateien untersucht.

Mittels der eben genannten Untersuchungen der einzelnen Javadocables werden die *"no Javadoc"* - Code Smells und zusätzlich die *"Paket benötigt Package-info"* sowie für typenlose Pakete - Code Smells erkannt.
Die restlichen Javadoc Code Smells werden mittels der Besuche der einzelnen Javadoc-Kommentare aus dem AST gefunden.
Dafür wird der Ablauf aus dem folgenden Aktivitätsdiagramm 3.5 verwendet. Zunächst wird überprüft, um welche Art von Javadocable es sich bei dem zum Javadoc-Kommentar gehörenden Javadocable handelt. Anschließend wird kontrolliert, ob diese Art von Javadocable von dem Detektor analysiert werden soll. Wenn es bei den Überprüfungen zu einer Übereinstimmung kommt, dann wird eine entsprechende Funktion ausgeführt. Dabei teilen sich die Executables sowie die Typen jeweils eine Funktion.
In dem Aktivitätsdiagramm unterscheiden sich die Aktionen, die diese Funktionen aufrufen, farblich von den anderen Aktionen. Diese Aktionen, werden im Folgenden noch explizit erläutert.

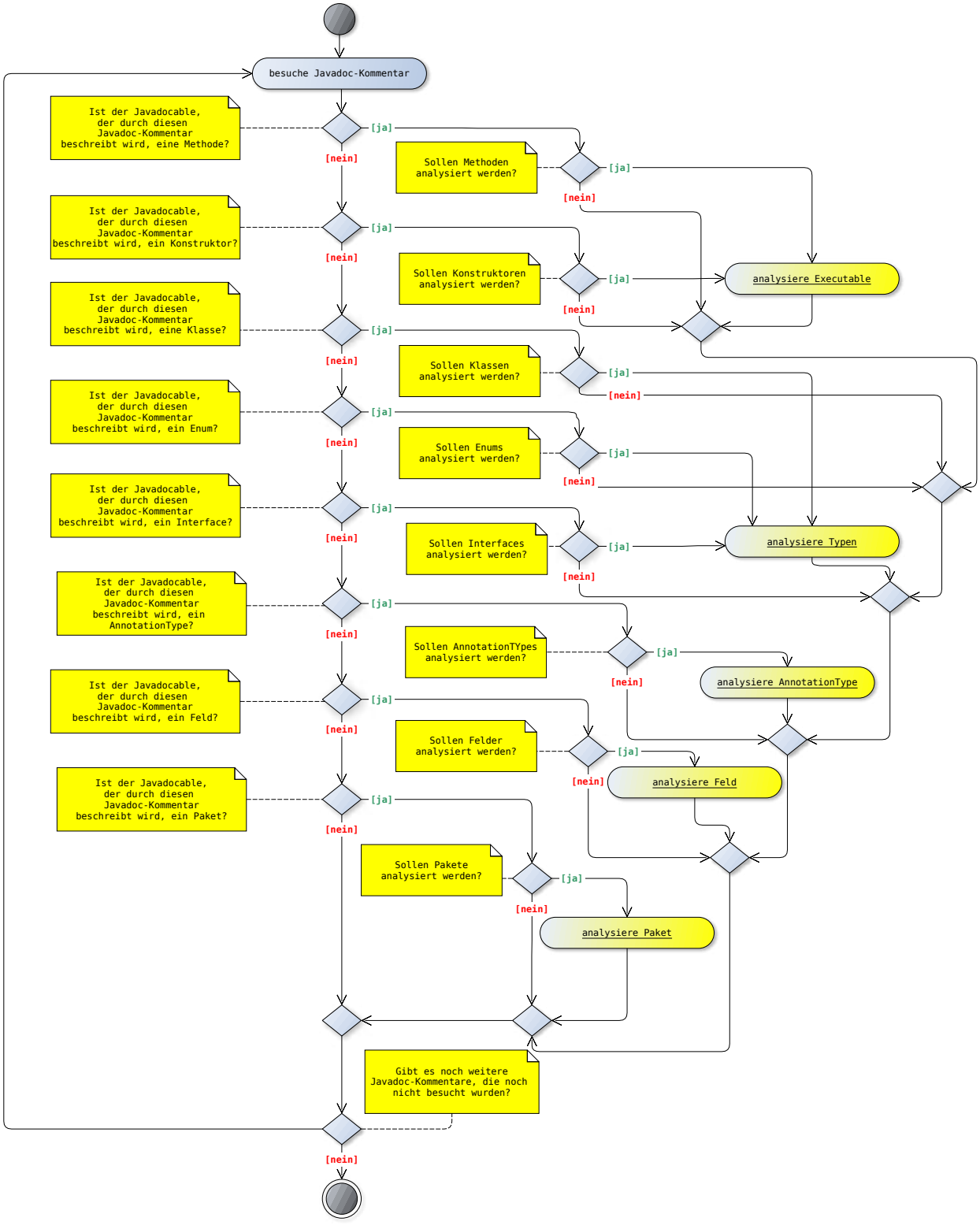


Abbildung 3.5.: Aktivitätsdiagramm - Ablauf der Besuche eines Javadoc-Kommentars.

Sobald der Javadoc-Kommentar zu einem Paket gehört und in der Konfiguration eingestellt wurde, dass der Detektor Pakete analysieren soll, durchläuft der Detektor die im Aktivitätsdiagramm 3.6 gezeigten Schritte.

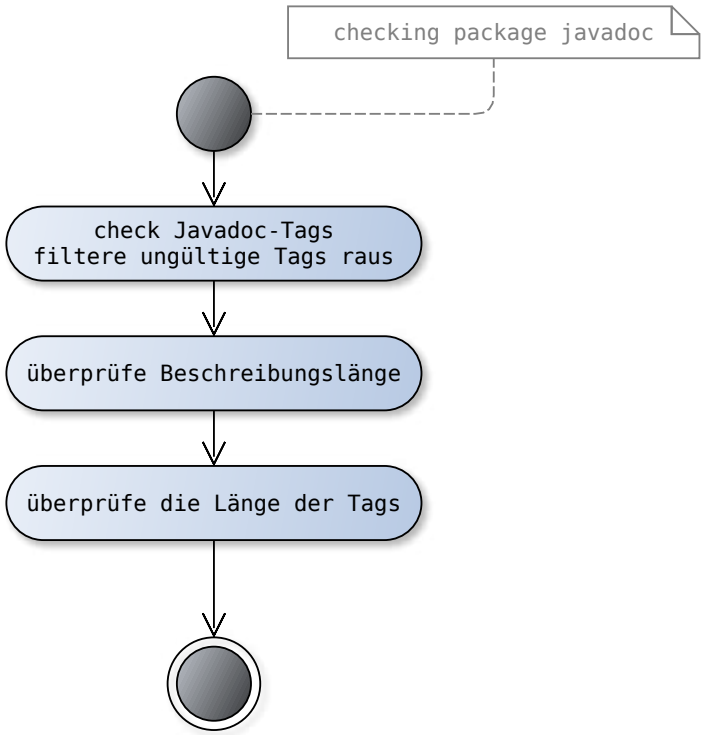


Abbildung 3.6.: Aktivitätsdiagramm - Ablauf der Analyse eines Paket Javadoc-Kommentars.

Als Erstes wird in diesem Durchlauf kontrolliert, ob alle vorhandenen Javadoc-Tags aus dem Javadoc-Kommentar für die Pakete zulässig sind. Dabei werden unzulässige Tags aussortiert und mittels eines Code Smells erkannt. Anschließend wird überprüft, ob die Längen der Javadoc-Beschreibung (Short-, Long-, Total-Description) die eingestellten Mindestlängen für Pakete erfüllen. Hierbei werden unzureichende Beschreibungen ebenfalls mit einem Code Smell versehen.
Hinweis: Im Folgenden ist zu beachten, dass die zulässigen Javadoc-Tags und die Beschreibungslängen immer von dem jeweiligen Javadocable abhängen.
Zum Schluss überprüft der Detektor sämtliche Beschreibungslängen der erlaubten Javadoc-Tags. Diejenigen, die nicht die konfigurierte Mindestlänge entsprechen, werden ebenfalls mit einem Code Smell gekennzeichnet.

Der Ablauf für die Untersuchung eines Javadoc-Kommentars einer Annotation ist im Aktivitätsdiagramm 3.7 zu sehen. Dabei funktioniert der Ablauf analog zu dem eines Paketes, mit dem Unterschied, dass vor den Überprüfungen der zulässigen Tags und der Längen, erst einmal eine Überprüfung des Zugriffsmodifikators stattfindet. Hierbei wird überprüft, ob der Zugriffsmodifikator des Javadocables von dem Javadoc-Detektor berücksichtigt werden soll. Wenn dies der Fall ist, funktioniert der Ablauf analog zu dem eines Paketes, dabei ist der eben genannte Hinweis zu beachten.

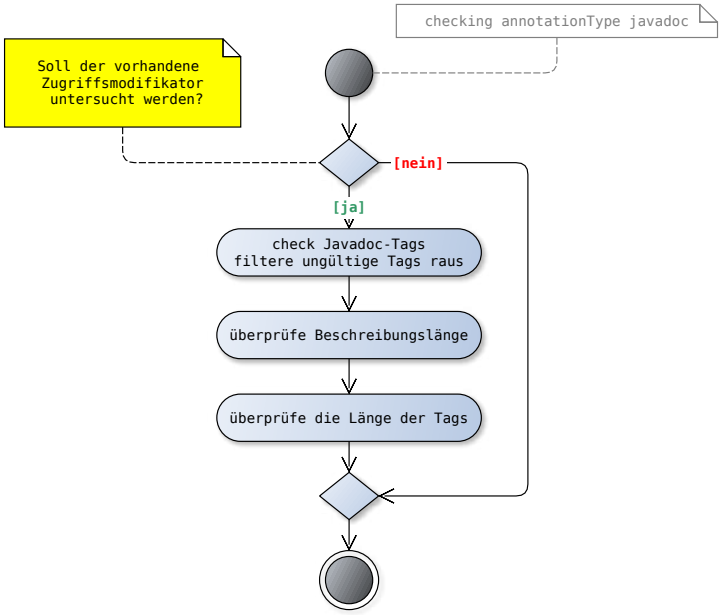


Abbildung 3.7.: Aktivitätsdiagramm - Ablauf der Analyse eines Annotation Javadoc-Kommentars.

Auf den Ablauf eines Annotation-Javadoc-Kommentars baut der Ablauf der Felder auf. Dieser wird durch die Deprecated-Überprüfung erweitert. Bei der Deprecated-Überprüfung wird überprüft, ob der Javadocable die Annotation *@Deprecated* sowie den dazugehörigen Javadoc-Tag *@deprecated* besitzt, falls es sich um einen veralteten Javadocable handelt. Sollte entweder die Annotation oder der Tag bei einem veralteten Javadocable fehlen, wird ein entsprechender Code Smell vermerkt. Das dazugehörige Aktivitätsdiagramm 3.8 ist im Folgenden zu sehen. Ansonsten funktioniert der Ablauf analog zu dem der Annotations.

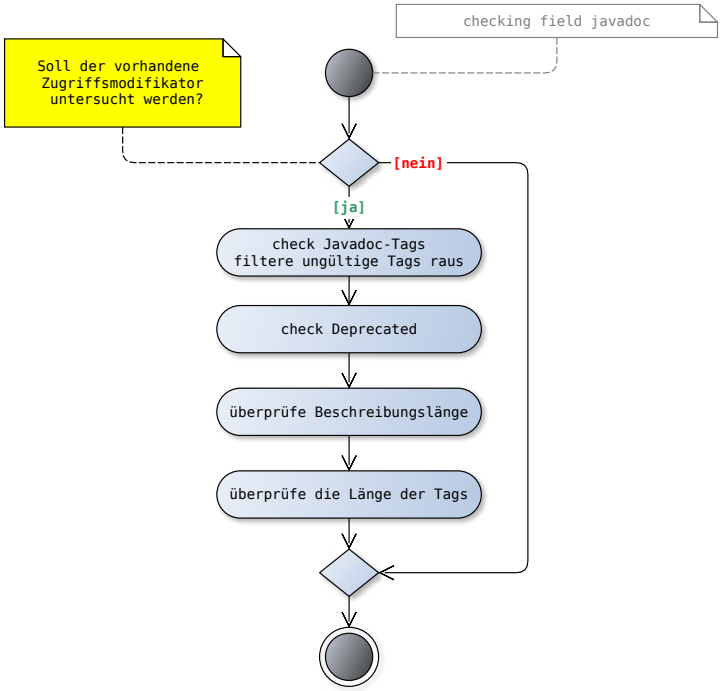


Abbildung 3.8.: Aktivitätsdiagramm - Ablauf der Analyse eines Feld Javadoc-Kommentars.

Die Javadoc-Kommentare der Executables durchlaufen in ihrem Ablauf unter anderem dieselben Aktionen wie ein Feld-Javadoc-Kommentar. Jedoch ist die Reihenfolge abgeändert und es finden zusätzliche Überprüfungen statt. Dabei werden die Exceptions sowie die Parameter und bei Methoden die Return-Funktion überprüft.
Bei der Überprüfung der Exceptions überprüft der Detektor, welche Exceptions von dem Executable oder von einem rekursiv aufgerufenen Executable geworfen werden kann. Diese Analyse funktioniert so lange rekursiv, bis der vorhandene Quellcode des Projektes verlassen wird, wie es bei der Nutzung von Funktionen aus Importen oder aus der Java API der Fall ist. Wenn der eben genannte Fall eintritt, dann kann lediglich anhand der vorhandenen Funktionsdeklaration überprüft werden, ob und welche Exceptions von der Funktion geworfen werden können. Außerdem überprüft dieser Schritt, ob alle dokumentierten Exceptions auch wirklich von dem Executable geworfen werden können. Sobald eine undokumentierte oder eine dokumentierte Exception, die nicht geworfen werden kann, gefunden wird, wird diese mit der Hilfe eines Javadoc Code Smells bemängelt.
Des Weiteren werden die durch einen *"try-catch"* - Block behandelten Exceptions herausgefiltert, da der Executable durch diese Behandlung mit der geworfenen Exception umgehen kann. Somit wird die Exception nicht mehr explizit geworfen, sondern abgefangen. [Ull12, S. 616]
Die Überprüfung der Return-Funktion setzt voraus, dass es sich bei dem gefundenen Executable um eine Methode handelt. Wenn dies der Fall ist, wird überprüft, ob die Methode einen Return-Wert besitzt und ob diese dokumentiert wurde. Sollte die Methode einen Return-Wert besitzen und dieser wurde nicht dokumentiert, oder sollte die Methode keinen Return-Wert besitzen und es wird dennoch einer im Javadoc-Kommentar beschrieben, dann werden entsprechende Code Smells vermerkt.
Die Funktion zur Überprüfung der Parameter funktioniert ähnlich zu der eben beschriebenen Funktion, die die Return-Funktion überprüft, nur dass statt des Return-Wertes Parameter überprüft werden. Dabei wird zum einen überprüft, ob alle Parameter aus der Parameterliste im Javadoc-Kommentar beschrieben wurden. Zum anderen wird überprüft, ob alle beschriebenen Parameter auch im dazugehörigen Javadocable existieren. Diese Überprüfung funktioniert ebenfalls für generische Parameter. Sollte es bei einer der Überprüfungen zu einem Fehler kommen, wird ein entsprechender Code Smell erkannt. Der Ablauf eines Executable-Javadoc-Kommentares kann im folgenden Aktivitätsdiagramm 3.9 betrachtet werden.

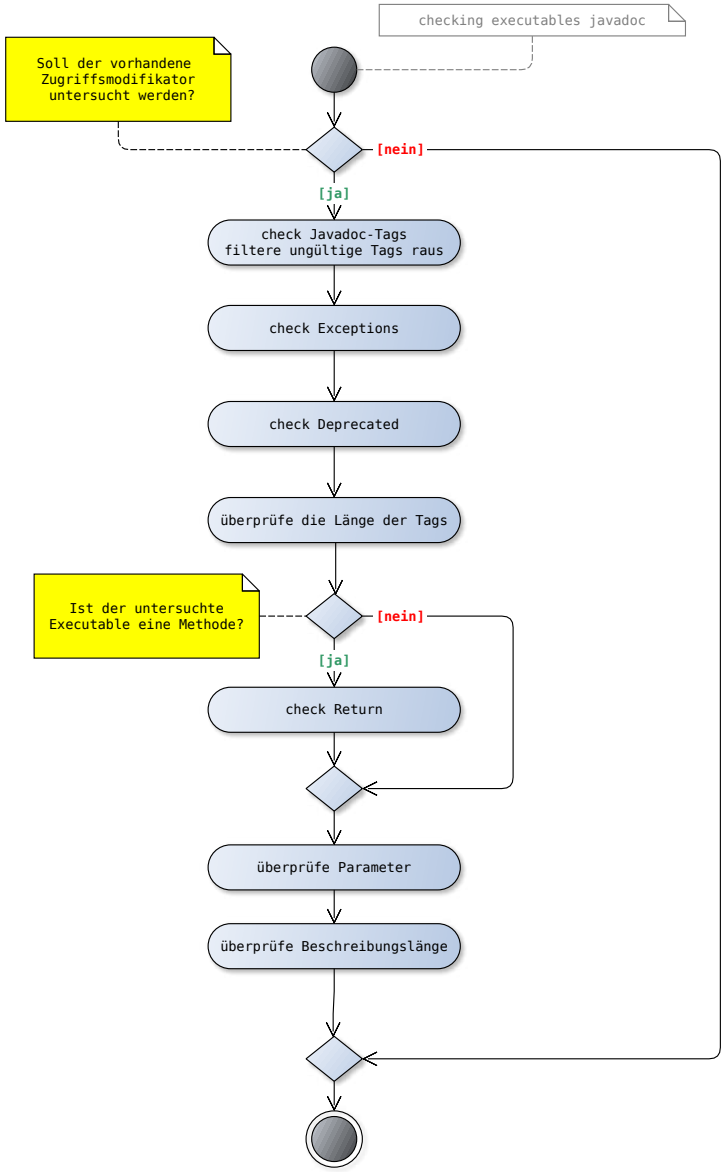


Abbildung 3.9.: Aktivitätsdiagramm - Ablauf der Analyse eines Executables Javadoc-Kommentars.

Die Überprüfung des Javadoc-Kommentars eines Typens beinhaltet in deren Durchlauf ebenfalls die Aktionen, die ein Feld in ihrem Ablauf durchläuft. Bei den Typen finden ebenfalls drei zusätzliche Überprüfungen statt. Diese finden nach der Überprüfung der zulässigen Javadoc-Tags statt. Wie aus dem Aktivitätsdiagramm 3.10 entnommen werden kann, handelt es sich bei diesen drei Überprüfungen um die Überprüfung des Autoren-Tags sowie die des Version-Tags und die der generischen Parameter. Dabei wird in der Überprüfung des Autoren- sowie Versions-Tag überprüft, ob ein solcher Tag im Javadoc-Kommentar vorhanden ist und ob eine Mindestlänge für den jeweiligen Tag konfiguriert wurde. Wenn der besagte Tag nicht im Javadoc-Kommentar beschrieben wurde, jedoch eine Mindestlänge für diesen Tag konfiguriert wurde und es sich bei den Typen um einen Top-Level-Typen handelt, dann wird ein entsprechender Code Smell erkannt. Die Überprüfung der Parameter ist lediglich für die generischen Parameter gedacht und funktioniert analog zu der Parameterüberprüfung der Executables.

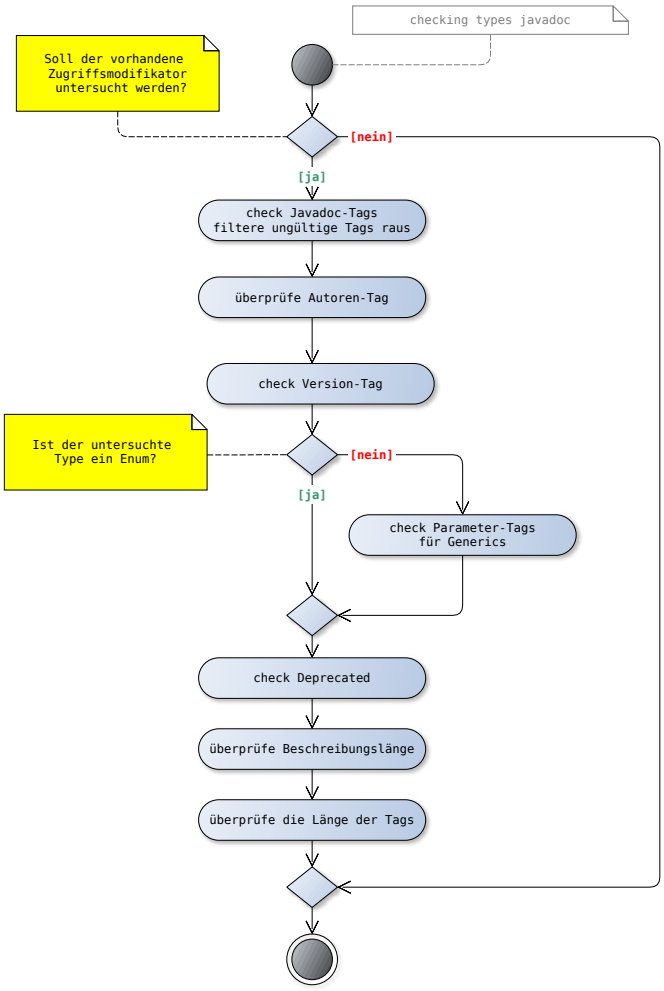


Abbildung 3.10.: Aktivitätsdiagramm - Ablauf der Analyse eines Typen Javadoc-Kommentars.

Die zusätzliche Deprecated-, Return, Parameter- oder Exception-Überprüfung findet nur statt, wenn eine Mindestlänge für den jeweiligen Javadoc-Tag konfiguriert wurde.

3.5. Javadoc Code Smell - Erkennung

Dieses Unterkapitel beschäftigt sich mit der Fragestellung, wann die im "Javadoc Code Smell - Arten" Kapitel genannten Code Smells auftreten und wie diese erkannt werden. Dabei werden die zusammengehörigen oder analog funktionierenden Code Smells jeweils zusammen in einem entsprechenden Unterbereich mit Beispielen beschrieben.

3.5.1. No Javadoc - Erkennung

Wie diese Art von Javadoc Code Smells erkannt werden, wurde bereits im vorherigen Unterkapitel ausführlich erläutert. Der folgende Beispielcode im Listing 3.2 enthält eine Klasse sowie ein Feld und eine Methode ohne Javadoc-Kommentar. Wenn der Detektor alle drei Javadocables analysieren soll und der Zugriffsmodifikator *"public"* aktiviert wurde, dann werden die drei Code Smells *"Class noJavadocClass has no javadoc."*, *"Field fieldWithoutJavadoc has no javadoc."* und *"Method methodWithoutJavadoc() has no javadoc."* erkannt.

```
1 public class noJavadocClass {
2   public int fieldWithoutJavadoc = 0;
3
4   public void methodWithoutJavadoc() {
5     ....
6   }
7 }
```

Listing 3.2: Ein Beispiel für die *"no Javadoc"* - Code Smells.

3.5.2. Unzulässiger Javadoc-Tag

```
1  /**
2    * Diese Methode zählt die Wörter des Textes.
3    * Dabei werden die zu erkennenden Wörter durch Leerzeichen oder Bindestriche getrennt.
4    * @param text enthält den Text, dessen Wörter gezählt werden sollen.
5    * @return die Anzahl der gezählten Wörter.
6    * @author Michel Krause
7  */
8  public int countWords(String text) {
9   ....
10 }
```

Listing 3.3: Ein Beispiel den Javadoc Code Smell *"Tag is not allowed"*.

Wenn der Javadoc-Detektor den im Listing 3.3 gezeigten Quellcode analysiert, unter der Voraussetzung, dass sämtliche Längen die konfigurierten Mindestlängen erfüllen und das

"public"-Methoden analysiert werden sollen, dann wird lediglich der Code Smell *"Tag author is not allowed for this javadocable."* erkannt. Dies geschieht mittels eines Vergleiches der im Javadoc-Kommentar vorhanden Javadoc-Tags mit den für diesen Javadocable erlaubten Tags. Welche Tags für welchen Javadocable zulässig sind, wurde bereits in dem Unterkapitel "Zugelassene Javadoc-Tags für die jeweiligen Javadocables" beschrieben.

3.5.3. Fehlender Autoren- oder Version-Tag - Erkennung

Sobald eine Mindestlänge für den Autoren- oder Version-Tag konfiguriert wird, wird jener Tag im Javadoc-Kommentar eines Top-Level-Types erwartet. Dies wird in der Dokumentation zum Thema "How to Write Doc Comments for the Javadoc Tool" im Kapitel "Order of Tags" genannt. [ORA04]
Wie im vorherigen Kapitel "Funktionsweise des Javadoc-Detektors" und eben beschrieben, werden die Code Smells *"Missing tag @author in the javadoc."* und *"Missing tag @version in the javadoc."* erkannt, sobald bei einem Top-Level-Typen der jeweilige Tag nicht beschrieben wurde, sofern eine Mindestlänge für den Tag konfiguriert wurde.
Im folgenden Listing 3.4 werden beide Code Smells für die Klasse *"foo"* erkannt. Unter der Voraussetzung, dass im Detektor die Mindestlängen für die beiden Tags sowie, dass *"public"* - Klassen untersucht werden sollen konfiguriert wurden. Dahingegen werden die eben genannten Code Smells für die innere Klasse *"bar"* nicht erkannt, da es sich bei der Klasse nicht um einen Top-Level-Typen handelt und somit diese Tags nicht benötigt werden.

```
1  /**
2   * Dies ist eine Beispiel Klassenbeschreibung.
3   * Die Klasse foo existiert nur zu Demonstrationszwecken. foo ist dabei eine Top-Level-Klasse.
4   */
5  public class foo {
6    ...
7    /**
8     * Dies ist eine Beispiel Klassenbeschreibung.
9     * Die innere Klasse bar existiert nur zu Demonstrationszwecken.
10    */
11   public class bar {
12     ...
13   }
14   ...
15 }
```

Listing 3.4: Ein Beispiel für die Javadoc Code Smells *"Missing tag @author ..."* und *"Missing tag @version ..."*.

3.5.4. Return - Erkennung

Zur Überprüfung, ob die Return-Funktion einer Methode richtig dokumentiert wurde, werden zwei verschiedene Javadoc Code Smells bereitgestellt. Zum einen der Smell *"Missing tag @return in the javadoc."* und zum anderen der Smell *"Javadoc contains @return, but the method has no return value."*. Der zuerst genannte Code Smell wird im Listing 3.5 demonstriert. In diesem Beispiel gibt die Methode *getRandomKeyword()* einen String zurück, diese Rückgabe wird jedoch nicht im Javadoc-Kommentar mittels des Javadoc-Tags *@return* dokumentiert. Dadurch wird vom Detektor der Code Smell *"Missing tag @return in the javadoc."* erkannt.

```
/**
 * Dies ist eine Beispiel Methodenbeschreibung.
 * Diese Methode gibt ein zufälliges Schlüsselwort der Java-Programmierung zurück.
 */
public String getRandomKeyword() {
  ...
}
```

Listing 3.5: Ein Beispiel für den Javadoc Code Smell *"Missing tag @return ..."*.

Im folgenden Listing 3.6 wird in dem Javadoc-Kommentar eine Rückgabe mittels des Tags *@return* beschrieben, die die Methode nicht leistet. Bei der Methode *"withParam(minLength)"* handelt es sich um eine sogenannte "void"-Methode. Diese "void"-Methoden können keinen Wert zurückgeben. Sobald der Detektor erkennt, dass es sich um eine void-Methode handelt, aber trotzdem ein Rückgabewert beschrieben wurde, dann wird der Code Smell *"Javadoc contains @return, but the method has no return value."* erkannt.

```
/**
 * Diese Methode konfiguriert die Mindestlänge für die Beschreibung der Parameter.
 * @param minLength beschreibt dabei die gewünschte Mindestlänge.
 * @return die aktuell konfigurierte Mindestlänge.
 */
public void withParam(int minLength) {
  tagParam = minLength;
}
```

Listing 3.6: Ein Beispiel für den Javadoc Code Smell *"Javadoc contains @return ..."*.

3.5.5. Parameter - Erkennung

Das folgende Listing 3.7 zeigt die Methode *"testMethod(String text)"* als Beispiel für die Erkennung von Javadoc Code Smells für Parameter. Diese Methode weist vier solcher Code Smells auf. Zweimal den Code Smell *"Javadoc contains @param"* für die Parameter *"<E>"* und *"number"*. Diese werden zwar im Javadoc-Kommentar beschrieben, aber existieren in der Methode nicht. Deshalb werden für die beiden Parameter jeweils ein Code Smell erkannt. Die erkannten Code Smells lauten *"Javadoc contains @param <E>, but this parameter does*

not exists.“ und *“Javadoc contains @param number, but this parameter does not exists.“*. Des Weiteren werden zweimal jeweils ein Code Smell für einen nicht beschriebenen Parameter erkannt. Zum einen ist dies der Code Smell *“Missing tag @param text in the javadoc.“* für den *“text“* - Parameter und zum anderen der Smell *“Missing tag @param <T> in the javadoc.“* für den generischen Parameter *“<T>“*.

```
1 /**
2  * Die Methode testMethod existiert nur zu Demonstrationszwecken.
3  * @param <E> existiert in dieser Methode nicht und ist deshalb ein Code Smell.
4  * @param number existiert in dieser Methode nicht und ist deshalb ebenfalls ein Code Smell.
5  */
6 public <T> void testMethod(String text) {
7   ...
8 }
```

Listing 3.7: Ein Beispiel für die Javadoc Code Smells *“Missing tag @param ...“* und *“Javadoc contains @param ...“*.

3.5.6. Deprecated - Erkennung

Zu der Deprecated-Erkennung gehören die Code Smells *“Missing tag @deprecated in the javadoc.“* und *“Javadoc contains @deprecated, but the annotation @Deprecated at the javadocable is missing.“*. Der erstgenannte Smell wird erkannt, wenn der untersuchte Javadocable an seiner jeweiligen Deklaration die Annotation *@Deprecated* besitzt und in seinem Javadoc-Kommentar nicht den dazugehörigen Javadoc-Tag *@deprecated* beschreibt. Ein Beispiel hierfür ist im Listing 3.8 zu sehen.

```
1 /**
2  * Die Methode testMethod existiert nur zu Demonstrationszwecken.
3  * Der gefundene Code Smell ist die fehlende Javadoc-Tag Beschreibung zu der Annotation-
       @Deprecated.
4  *
5  */
6 @Deprecated
7 public void testMethod() {
8   ...
9 }
```

Listing 3.8: Ein Beispiel für den Javadoc Code Smell *“Javadoc contains @deprecated ...“*

Hingegen wird der Code Smell *"Javadoc contains @deprecated, but the annotation @Deprecated at the javadocable is missing."* erkannt, wenn die eben genannte Annotation vor der Deklaration fehlt und der Javadoc-Kommentar den *@deprecated* - Tag beschreibt. Im Listing 3.9 wird hierzu ein Beispiel gegeben.

```
1 /**
2  * Die Methode testMethod existiert nur zu Demonstrationszwecken.
3  * Der gefundene Code Smell ist die fehlende @Deprecated-Annotation vor der Methodendeklaration.
4  * @deprecated Methode ist aufgrund der neuen Strukturierung aus Version 1.3 nicht mehr zu
       verwenden, statt dessen soll die Methode xy verwendet werden.
5  */
6 public void testMethod() {
7   ...
8 }
```

Listing 3.9: Ein Beispiel für den Javadoc Code Smell *"Missing tag @deprecated ..."*

3.5.7. Exception - Erkennung

Der Javadoc-Detektor bietet für die Erkennung von Exception Code Smells, die folgenden beiden Code Smells *"Missing tag @throws ..."* und *"Javadoc contains @throws/@exception ..."* an. Dabei wird der erstgenannte Code Smell erkannt, sobald eine Exception nicht mittels des Javadoc-Tags *@throws* oder *@execption* dokumentiert wurde. Die zu überprüfenden Exceptions werden dabei von dem analysierten Executable oder von einem aufgerufenen Executable geworfen.
Daher würde der Detektor bei einer Überprüfung des Beispiels aus dem Listing 3.10 für die Exceptions *"ParseException"* und *"IllegalArgumentException"* den Code Smell erkennen, dass diese Exceptions nicht dokumentiert wurden.
Außerdem wird in dem Beispiel die *"ClassCastException"* dokumentiert. Diese wird jedoch weder von der Methode *"testMethod(File file, String text)"* noch von den aufgerufenen Methoden *"checkFile(File file)"* oder *"parseText(String text)"* geworfen. Deshalb kann die *"ClassCastException"* nicht von der Methode *"testMethod(File file, String text)"* geworfen werden. Aus diesem Grund wird für diese Exception der zweitgenannte Code Smell *"Javadoc contains @throws/@exception ClassCastException, but this does not thrown by this or a called executable."* erkannt.
Neben den eben gezeigten Fall, dass die dokumentierte Exception exakt die geworfene Exception beschreibt, gibt es einen weiteren Fall aufgrund der Vererbung von Klassen beziehungsweise der Exceptions. Dabei kann eine dokumentierte Exception eine Exception ihrer Superklasse beschreiben. Dafür muss jedoch die dokumentierte Exception eine Subklasse der geworfenen Exception (Superklasse) sein. Ein Beispiel hierzu wird ebenfalls im Listing 3.10 gegeben. Die Methode *"testMethod(File file, String text)"* kann eine *"IOException"* werfen und in ihrem Javadoc-Kommentar wird eine *"FileNotFoundException"* beschrieben. Damit wird die *"IOException"* hinreichend beschrieben, da eine *"FileNotFoundException"* eine Unterexception von ihr ist.
Aufgrund der Vererbung ist der umgekehrte Fall jedoch nicht möglich. Das bedeutet, dass

eine dokumentierte Exception, die eine Superklasse von der geworfene Exception (Subklasse) ist, diese nicht beschreiben kann und darf. Damit ist für das eben genannte Beispiel gemeint, dass eine *"IOException"* keine *"FileNotFoundException"* dokumentieren kann, da die *"FileNotFoundException"* eine Spezialisierung der *"IOException"* ist. [PP19][Ull12, S. 547-548]

Um zu erkennen, ob die dokumentierte Exception eine Subklasse der geworfenen Exception ist, muss das dazugehörige Spoon-Element im AST gefunden werden. Mittels des Spoon-Elements kann die Vererbungshierarchie zu der geworfenen Exception überprüft werden. Für diese Überprüfungen stellt die Spoon-API Funktionen bereit. Jedoch werden für diese Überprüfungen der vollqualifizierte Name der dokumentierten Exception benötigt. Der vollqualifizierte Name beinhaltet dabei den Namen der Exception sowie ihre deklarierenden Pakete.
Zur Ermittlung des vollqualifizierten Namens der Exception stellt der Javadoc-Detektor drei verschiedene Möglichkeiten bereit.

- **1. Möglichkeit:** Bei dieser Möglichkeit versucht der Detektor den vollqualifizierten Namen der Exception über die Imports des Typens zu finden. Der Typ ist dabei derjenige Typ, der den Executable implementiert. Der Executable ist dabei der Besitzer des Javadoc-Kommentars, der die dokumentierte Exception beinhaltet.

- **2. Möglichkeit:** Bei dieser Möglichkeit wird überprüft, ob die dokumentierte Exception in demselben Paket wie die geworfene Exception deklariert ist.

- **3. Möglichkeit:** Als letzte Möglichkeit versucht der Detektor mit der Hilfe Klasse *"JavaExceptionMap"* den vollqualifizierten Namen der dokumentierten Exception zu erhalten. Wie genau das funktioniert, wird im nachfolgenden Unterkapitel "JavaExceptionMap" beschrieben.

```java
/**
 * Die Methode testMethod existiert nur zu Demonstrationszwecken.
 * Die Exceptions ParseException und IllegalArgumentException, die von dieser Methode geworfen
    werden können werden nicht beschrieben. Dies stellt jeweils ein Code Smell dar.
 * @param file ist die Datei, die überprüft werden soll.
 * @param text ist der Text, der geparst werden soll.
 * @throws FileNotFoundException wenn die gegebene Datei nicht gefunden werden kann.
 * @exception ClassCastException diese Exception kann von der Methode, oder einer aufgerufenen
    Methode nicht geworfen werden und ist deshalb auch ein Code Smell.
 *
 */
public void testMethod(File file, String text) throws IOException, ParseException {
  if (text == null || file == null) {
    throw new IllegalArgumentException("text und file darf nicht null sein.");
  }
  .....
  Boolean check = checkFile(file);
  String parsedText = parseText(text);
  .....
}

/**
 * Die Methode checkFile existiert nur zu Demonstrationszwecken.
 * @param file die Datei, die überprüft werden soll.
 * @throws FileNotFoundException wenn die gegebene Datei nicht gefunden werden kann.
 *
 */
private Boolean checkFile(File file) throws IOException {
  ....
}

/**
 * Die Methode parseText existiert nur zu Demonstrationszwecken.
 * @param text der geparst werden soll.
 * @throws ParseException wenn ein Fehler bei dem parsen auftritt.
 *
 */
private String parseText(String text) throws ParseException {
  ....
}
```

Listing 3.10: Ein Beispiel für die Javadoc Code Smells *"Missing tag @throws ..."* und *"Javadoc contains @throws/@exception ..."*

3.5.7.1. JavaExceptionMap

Die JavaExceptionMap ist eine Klasse, die sämtliche öffentlichen Exceptions und Errors der Java API (Version 8) beinhaltet.
Mittels einer Funktion, die als Eingabe den einfachen Namen der Exception oder des Errors erhält, wird der vollqualifizierten Namen der Exception beziehungsweise des Errors ermittelt.
Beispiel:
Eingabe: *"IllegalArgumentException"* -> Ausgabe: *"java.lang.IllegalArgumentException"*

Die Liste der Exceptions und Errors, auf den die JavaExceptionMap basiert, wurde von der Programming.Guide-Webseite[2] entnommen.

3.5.8. Erkennung der Mindestlänge für die Tag-Beschreibung

Für das Beispiel im Listing 3.11 gelten die Javadoc-Tag-Konfigurationen aus dem oben genannten Konfigurationsbeispiel aus dem Kapitel "Konfigurationsmöglichkeiten". Das Listing umfasst dabei nicht alle Javadoc-Tags, jedoch funktioniert das Erkennen der Tag-Längen für jeden Tag identisch. Dabei wird je nach Konfiguration die Wörter oder die Zeichen gezählt. Die gezählten Wörter/Zeichen umfassen dabei immer die Beschreibung des Tags. Parameter zählen nicht zu der Beschreibung. Um dies auf das unten stehende Beispiel anzuwenden, wäre das zum Beispiel für den *@return* - Tag der Text *"repeated word."* und für den *@param c* - Tag der Text *"repetead Char."*. Dabei würde die Zählfunktion für den *@return* - Tag zwei Wörter oder 14 Zeichen und für den *@param c* - Tag ebenfalls zwei Wörter oder 19 Zeichen zählen. Wie schon mehrfach beschrieben, werden die verschiedenen Wörter dabei durch ein oder mehrere Leerzeichen oder Bindestriche getrennt. Sobald die gezählte Tag-Länge nicht die konfigurierte Mindestlänge des jeweiligen Tags erfüllt, wird ein entsprechender Code Smell erkannt. Die entsprechenden Code Smells wurden bereits im Kapitel "Javadoc Code Smell - Arten" in der Oberkategorie *"Too short tag description"* beschrieben.

```java
/**
     * Dies ist ein Beispiel für zu kurze Tag-Beschreibungslängen.
     * Das Vorgehen funktioniert zu jedem nicht genannten Tag analog zu den Folgenden.
     * @param c repeated character.
     * @param repeatNumber how often repeated.
     * @return repeated word.
     * @throws IllegalArgumentException by wrong number.
     * @see
     * @since
     * @deprecated too short
     */
    @Deprecated
    public String charRepeater (char c, int repeatNumber) {
        String repeatedChar = "";
        if (repeatNumber < 1) {
            throw new IllegalArgumentException("no repeat possible");
        }
        for (int i = 0; i < repeatNumber; i++) {
            repeatedChar += c;
        }
        return repeatedChar;
    }
```

Listing 3.11: Ein Beispiel für die Javadoc Code Smells zur Erkennung der verschiedenen Tag-Längen.

[2]https://programming.guide/java/list-of-java-exceptions.html

3.5.9. Erkennung der Mindestlänge für die Beschreibung

Die Erkennung der Code Smells für die Mindestlänge der Beschreibungen funktioniert analog zu der Erkennung der Tag-Längen. Mit dem Unterschied, dass hierbei der jeweilige Text der Short-, Long- oder Total-Description ausgewertet wird.
Im Listing 3.12 wird hierfür erneut das Beispiel von Wikipedia[3] verwendet. Für die Short-Description würden vier Wörter sowie 27 Zeichen, für die Long-Description 24 Wörter und 155 Zeichen und für die Total-Description 28 Wörter und 182 Zeichen gezählt werden. Sobald eine dieser gezählten Längen nicht die konfigurierte Mindestlänge für die Beschreibung erfüllt, wird ein entsprechender Code Smell erkannt. Durch dieses Beispiel werden die Code Smells *"Short-description of this javadoc is too short."* sowie *"Long-description of this javadoc is too short."* und *"Total-description of this javadoc is too short."* abgedeckt.

```
1  /**
2   * Short one line description.
3   * Longer description. If there were any, it would be
4   * here.
5   * And even more explanations to follow in consecutive
6   * paragraphs separated by HTML paragraph breaks.
7   *
8   * @param  variable Description text text text.
9   * @return Description text text text.
10  */
11 public int methodName (...) {
12     // method body with a return statement
13     String test = "Hallo Welt!";
14      throw new IllegalArgumentException();
15 }
```

Listing 3.12: Ein Beispiel für die Javadoc Code Smells zur Erkennung der verschiedenen Beschreibungslängen.

Der fehlende Tag dieser Kategorie *"'No description existing in this javadoc."* wird im nachfolgenden Listing 3.13 demonstriert. Sobald der Detektor einen Javadoc-Kommentar mit keiner Javadoc-Beschreibung vorfindet, wird dieser Smell erkannt. Wenn der unten stehende Javadoc-Kommentar Javadoc-Tags dokumentieren würde, würde dennoch der Smell erkannt werden.

```
1  /**
2   *
3   */
4  public void testMethod(...) {
5  }
```

Listing 3.13: Ein Beispiel für die Javadoc Code Smells zur Erkennung der verschiedenen Beschreibungslängen.

[3]https://en.wikipedia.org/wiki/Javadoc

3.5.10. Fehlende Package-info - Erkennung

Anhand des folgenden Listings 3.14 werden die beiden Code Smells *"Missing package-info for the package"* und *"Missing package-info for the typeless package"* erklärt. Dabei wird angenommen, dass das Paket *"javadoc"* ein typenloses Paket ist. Dieses Paket deklariert demnach nur weitere Unterpakete, wie zum Beispiel das Paket *"detection"*. Des Weiteren wird für beide Pakete angenommen, dass sie keine Package-info-Datei besitzen.
Im Folgenden wird angenommen, dass die Konfigurationen "Paket benötigt Package-info" sowie "Typenlose Pakete benötigen Package-info" aktiviert wurden. Sobald der Javadoc-Detektor die Klasse *"foobar"* hinsichtlich der Existenz seines Javadoc-Kommentars untersucht, wird ebenfalls untersucht, ob das Paket, welches die Klasse *foobar* deklariert, eine Package-info-Datei bestizt. In diesem Fall ist dies das Paket *"detection"*. Wenn dies Paket keine Package-info besitzt, dann wird der Code Smell *"Missing package-info for the package detection."* erkannt.
Anschließend untersucht der Detektor das Paket, das *"detection"* deklariert. In diesem Fall, ist dass das Paket *"javadoc"*. Falls das Paket *"javadoc"* keine Package-info-Datei besitzt, wird der Code Smell *"Missing package-info for the typeless package javadoc."* erkannt.

```
1 package javadoc.detection;
2
3 /**
4  * Die Klasse foobar existiert nur zu Demonstrationszwecken.
5  * Sie wird im Paket detection deklariert.
6  */
7 public class foobar {
8   ...
9 }
```

Listing 3.14: Ein Beispiel für die Javadoc Code Smells *"Missing package-info for the package ..."* und *"Missing package-info for the typeless package ..."*.

4. Evaluation

In diesem Kapitel wird die Evaluation zu der durchgeführten Javadoc Code Smell - Analyse mittels des, im vorherigen Kapitel beschriebenen, Javadoc-Detektors behandelt. Dazu wird zuerst erläutert wie die benötigten Daten für die Forschungsfragen erhoben und gesammelt wurden. Anschließend werden die drei Forschungsfragen, die in der Einleitung definiert wurden, mittels der erhobenen Daten beantwortet.

4.1. Datenerhebung

In diesem Abschnitt wird das Thema der Datenerhebung für die Auswertung der durchgeführten Analyse beschrieben. Um die drei Forschungsfragen dieser Bachelorarbeit beantworten zu können, werden die ausgewerteten Javadoc Code Smells der analysierten Projekte benötigt.
Zu beachten ist hierbei, dass von den untersuchten Projekten nur der Quellcode inklusive Dokumentation des eigentlichen Produktes analysiert wird. Die mitgelieferten Testklassen sowie Beispiele werden in der Analyse nicht berücksichtigt. Testklassen und Beispiele werden ausgeschlossen, da sie größtenteils keine Javadoc-Dokumentation besitzen und dadurch die Analyse verfälschen würden. Die zu analysierenden Projekte sind dieselben Projekte, die bereits Artur Bosch in seiner Masterarbeit zum Thema "Priorisierung von Code Smells" [Bos18] und Dominique Schulz in seiner Bachelorarbeit zum Thema "Evolution von Code Smells" [Sch19] verwendet haben. Jedoch waren einige von den ursprünglichen 200 Projekten nicht mehr verfügbar oder ließen sich nicht mehr analysieren. Dominique Schulz erstellte für seine Bachelorarbeit eine Liste mit den Pfaden, in denen der für die Analyse relevante Quellcode der Projekte zu finden ist. Diese Liste wurde auch für die Analyse, die dieser Evaluation zugrunde liegt, verwendet. Im Anhang B ist die Liste der analysierten Projekte zu finden.
Für die Auswertung der Daten der analysierten Projekte wurde eine Klasse namens *"Evaluator"* in der Programmiersprache Java implementiert. Die Klasse *"Evaluator"* bereitet dabei die gelieferten Daten (Javadoc Code Smells) des Javadoc-Detektors auf, sodass mittels der aufbereiteten Daten die Forschungsfragen beantwortet werden können.
Um die erste Forschungsfrage *RQ1* beantworten zu können, werden Daten von den Projekten zu verschiedenen Zeitpunkten benötigt, um den Verlauf ihrer Javadoc Code Smells darstellen zu können. Insgesamt wurden hierfür 177 Projekte hinsichtlich ihrer Historie analysiert und ausgewertet. Der Evaluator fasst hierfür die Daten, wie im Kapitel "Forschungsfragen" beschrieben, in Oberkategorien zusammen. Dies wird in dem Unterkapitel "RQ1 - Die Evaluation der Javadoc Code Smell Historie der Projekte" näher erläutert. Außerdem erstellt der Evaluator für jedes analysierte Projekt eine CSV-Datei, in der für jeden untersuchten Zeitpunkt des Projektes die Oberkategorien sowie eine Ordnungszahl, die den Zeitpunkt re-

präsentiert, und die Anzahl der vorhandenen Javadocables geschrieben sind.
Dahingegen wird für die Beantwortung der Forschungsfragen *RQ2* und *RQ3* lediglich der letzte verfügbare Stand der jeweiligen Projekte benötigt. Für diese Forschungsfragen wurden 182 Projekte analysiert und ausgewertet. Der Evaluator legt für diese Auswertungen verschiedene CSV-Dateien an.
Für die Forschungsfrage *RQ2* werden dabei viele verschiedene Dateien angelegt. Zum einen wird für jeden Javadoc Code Smell eine CSV-Datei angelegt, in der beschrieben ist, wie oft es in dem jeweiligen Projekt zu einem Verstoß gegen diesen Smell gekommen ist und anschließend wird aufgezählt, wie häufig dieser Smell in den jeweiligen Javadocables vorkam. Des Weiteren werden für jeden der acht Javadocables CSV-Dateien angelegt, in denen alle gefundenen Javadoc Code Smells in dem jeweiligen Javadocable auflistet werden. Die letzte Datei, die für diese Forschungsfrage angelegt wird, beinhaltet die komplette Sicht auf die verschiedenen Projekte. In dieser Datei ist für jedes Projekt aufgelistet, wie oft die einzelnen Javadoc Code Smells vorkamen. Zusammenfassend stellt der Evaluator für diese Forschungsfrage die verschiedenen Javadoc Code Smells aus verschiedenen Perspektiven dar.
Dahingegen legt der Evaluator für die Forschungsfrage *RQ3* nur eine CSV-Datei an, in der für jedes Projekt die Anzahl der vorhandenen Javadocables, die Anzahl der von Code Smells befallen Javadocables sowie die Anzahl der nicht befallenen Javadocables und dasselbe für jeden einzelnen Javadocable aufgeführt wird. Dabei bezieht sich der Befall und das Freisein von Code Smells nur auf die Javadoc Code Smells.
Die Differenz zwischen der Anzahl der untersuchten Projekte für die verschiedenen Forschungsfragen liegt daran, dass es bei fünf Projekten während der Analyse der Historie zu Problemen gekommen ist. Diese Probleme sind auf uncompilierbare Projektstände zurückzuführen.
Für die Analyse der Projekte wurde die bereits im Listing 3.1 im Kapitel “Konfigurationsmöglichkeiten“ erläuterte Konfiguration verwendet.

4.2. Forschungsfragen

Die Forschungsfragen, die dieser Bachelorarbeit zugrunde liegen, wurden bereits im Kapitel 1.2 formuliert. In den folgenden Unterkapiteln werden diese drei Forschungsfragen beantwortet.

4.2.1. RQ1 - Die Evaluation der Javadoc Code Smell Historie der Projekte

Diese Forschungsfrage widmet sich der Frage, wie sich die analysierten Projekte über die Zeit hinweg in Bezug auf Javadoc Code Smells entwickeln. Wie bereits in der Datenerhebung und in dem Abschnitt Forschungsfragen in der Einleitung beschrieben, werden verschiedene Daten von den Projekten benötigt, um diese Forschungsfrage beantworten zu können. Neben der Anzahl der Javadocables und der Anzahl der gesamten Javadoc Code Smells zu einem bestimmten Zeitpunkt, werden mehrere Code Smells in Oberkategorien zusammengefasst und erhoben. Diese Oberkategorien werden dabei wie in dem Kapitel "Javadoc Code Smell - Arten" beschrieben zusammengefasst. Mit dem Unterschied, dass hierbei die Oberkategorien *"Too short description"* und *"Too short tag description"* zur allgemeinen Oberkategorie *"short Description"* zusammengefasst wird. Diese Zusammenfassung erfolgt aufgrund dessen, dass beide Unterkategorien an Mindestlängen bemessen werden und die Reduzierungsmaßnahmen der jeweiligen Code Smells äquivalent sind.
Somit werden die Oberkategorien für *"short Description"* sowie *"no Javadoc"*, *"missing Tag"*, *"missing Tag-Requirement"*, *"unallowed Tag"* und *"missing Package-info"* erhoben.
Um nun Aussagen über den Verlauf der Javadoc Code Smells treffen zu können, müssen die erhobenen Daten der analysierten Projekte einer weiteren Analyse unterzogen werden. Die hierfür verwendete Analyse nennt sich Mann-Kendall-Trendanalyse beziehungsweise Mann-Kendall-Trend-Test.
Durch diesen Test lassen sich Aussagen darüber treffen, ob die untersuchten Code Smells einen aufsteigenden, absteigenden oder keinen Trend besitzen. Kein Trend bedeutet dabei jedoch nicht zwangsläufig, dass die Anzahl der untersuchten Code Smells gleichbleibend ist, sondern kein Trend ist ebenfalls das Ergebnis, wenn die untersuchten Werte zu stark schwanken und somit kein monotoner Trend existiert. Die Nullhypothese des Tests nimmt dabei die Vermutung an, dass kein monotoner Trend existiert. Dahingegen nimmt die alternative Hypothese an, dass ein aufsteigender oder absteigender Trend existiert. [Ste16, Zai12]
Die Code Smells besitzen einen monoton aufsteigenden Trend, wenn die Anzahl der gefunden Javadoc Code Smells über die Zeit hinweg stetig steigen. Absteigender Trend bedeutet, dass die Javadoc Code Smells des Projektes über die Zeit hinweg stetig reduziert wurden. Dies gilt analog für die Javadocables. Für diese Bachelorarbeit wurde die Implementation des Mann-Kendall-Trend-Tests von Michael Schramm[1] verwendet. Schramm stellte dabei seine Implementierung des Mann-Kendall-Tests, die auf den ursprünglichen Code von Sat Kumar Tomer basieren, auf Github bereit. Diese Implementierung erwartet als Eingabe einen Vektor von Daten und einen Signifikanzniveau. Dafür wurden bei den Tests, das von Schramm

[1] https://github.com/mps9506/Mann-Kendall-Trend

eingestellte, Signifikanzniveau von 0.05 beibehalten.
Das Signifikanzniveau beschreibt dabei, wie hoch die Wahrscheinlichkeit des Risikos ist, eine falsche Entscheidung bezüglich der Existenz des Trends zu treffen. [Hem11]

In den nachfolgenden Tabellen sind mit Code Smells nur die Javadoc Code Smells gemeint. Mittels der Tabelle 4.1 ist deutlich zu erkennen, dass sich die Javadocables und die Javadoc Code Smells der analysierten Projekte, größtenteils proportional zueinander verhalten. Dieses Verhalten wird in der Tabelle 4.1 durch die gelb hinterlegten Zahlen repräsentiert. Dabei besitzen 84,18% der Projekte einen aufsteigenden, lediglich 3,39% einen absteigenden und 12,43% keinen Javadoc Code Smell Trend. Wenn das Projekt einen aufsteigenden Javadocables-Trend besitzt, dann war es in 145 Projekten beziehungsweise zu 97% ebenfalls der Fall, dass die Code Smells auch einen aufsteigenden Trend vorwiesen. Jedoch gab es vier Projekte mit einem aufsteigenden Javadoc Code Smell Trend und keinem Javadocable-Trend. Das bedeutet, das im Laufe der Zeit die Dokumentation der Javadoc-Kommentare schlechter wurden. Dies lässt sich zum Beispiel dadurch erklären, dass die Methoden komplexer und nicht mit dokumentiert wurden. In der Tabelle 4.1 werden diese vier Projekte durch einen roten Hintergrund markiert. Insgesamt wurden lediglich nur sechs Projekte mit absteigenden Code Smell Trend gefunden, das bedeutet, dass bei den Projekten die Javadoc-Kommentare besser dokumentiert wurden. Allerdings wiesen vier der sechs Projekte ebenfalls einen absteigenden Javadocables-Trend auf. Dies könnte ebenfalls eine Variante sein, weshalb sich die Javadoc Code Smells reduziert haben, denn wenn der Javadocable inklusive Kommentar entfernt wird, entfallen ebenfalls die dazugehörigen Code Smells. Allerdings sind zwei dieser sechs Projekte sehr interessant, denn im Laufe der Zeit wurden die Javadoc Code Smells reduziert, obwohl die Javadocables anstiegen oder kein Trend vorweisen konnten. Diese beiden Fälle sind in der Tabelle 4.1 grün hinterlegt. Das Projekt mit den ansteigenden Javadocables ist das Projekt *“JTK von MinesJTK“*, dieses Projekt wurde scheinbar stetig gut dokumentiert, denn dessen Commit-Bezeichnungen wiesen kaum Verbesserungen an den Javadoc-Kommentaren auf. Ebenso gilt dasselbe für das Projekt *“Tape von Square“*, mit den gleichbleibenden beziehungsweise stark schwankenden Javadocables. Bei beiden Projekten waren lediglich circa zwei Prozent aller Commit-Bezeichnungen auf Javadoc-Kommentar-Verbesserungen zurückzuführen, das bedeutet, dass die Javadoc-Kommentare nebenher verbessert wurden und keine expliziten Commits erhielten.

Javadocables / Code Smells	Trend aufsteigend	Trend absteigend	kein Trend
Trend aufsteigend	145	0	4
Trend absteigend	1	4	1
kein Trend	2	0	20

Tabelle 4.1.: Übersicht der Mann-Kendall-Trendanalyse aus der Sicht der Javadocables auf die vorhandenen Javadoc Code Smells.

In der nachfolgenden Tabelle 4.2 werden die Projekte, die keinen Javadoc Code Smells Trend aufweisen, dargestellt. Dabei ist interessant, dass es kein Projekt gibt, das dennoch einen aufsteigenden Trend für die unzulässigen Javadoc-Tags besitzt. Des Weiteren ist hierbei sehr interessant das sechs von den 22 Projekten einen absteigenden Trend für die Smells der Kategorie “Package-info“ besitzen, das bedeutet, dass diese Projekte mehr Package-Info-Dateien für ihre Pakete, über die Zeit hinweg, angelegt haben.

Code Smells / kein Trend	short Description	no Javadoc	missing Tag	missing Tag-Requirement	unallowed Tag	missing Package-info
Trend aufsteigend	3	2	3	2	0	2
Trend absteigend	3	2	2	2	2	6
kein Trend	16	18	17	18	20	14

Tabelle 4.2.: Übersicht der Mann-Kendall-Trendanalyse aus der Sicht der Code Smells auf die Projekte ohne vorhandenen Code Smell Trend.

Die nachfolgende Tabelle 4.3 beinhaltet einige interessante Fälle, obwohl in dieser Tabelle die Projekte mit einem ansteigenden Trend für die Javadoc Code Smells dargestellt werden. So gibt es zum Beispiel drei Projekte, die einen absteigenden Trend für die Smells des fehlenden Javadoc-Kommentars besitzen, somit wurden in diesen Projekten, über die Zeit hinweg, stetig Javadoc-Kommentare hinzugefügt. Anhand der Commit-Historie dieser Projekte ließ sich jedoch keine explizite Nachbesserung der Javadoc-Kommentare feststellen, woraus sich erschließen lässt, dass die Javadoc-Kommentare nebenher angelegt wurden und keine eigenen Commits erhielten. Des Weiteren existieren sechs Projekte, die einen absteigenden Trend für die fehlenden Javadoc-Tags in den Kommentaren besitzen. Das bedeutet, dass diese Projekte explizit ihre Javadoc-Kommentare im Hinblick auf fehlende Tags verbessert haben.
Außerdem ist es sehr interessant zu sehen, dass die Smells für die unzulässigen Tags in der Regel nicht so häufig mit ansteigen, sondern eher gleich bleiben beziehungsweise schwanken. Des Weiteren ist ebenfalls interessant, dass die fehlenden Tag-Bedingungen in der Regel nur in minimal mehr Projekten mit ansteigen, als in den Projekten in denen diese Smells gleich bleiben oder schwanken.

Code Smells / auf-steigender Trend	short Description	no Javadoc	missing Tag	missing Tag-Requirement	unallowed Tag	missing Package-info
Trend aufsteigend	132	139	133	77	34	89
Trend absteigend	3	3	6	20	21	25
kein Trend	14	7	10	52	94	35

Tabelle 4.3.: Übersicht der Mann-Kendall-Trendanalyse aus der Sicht der Code Smells auf die Projekte mit aufsteigendem Code Smell Trend.

In der Tabelle 4.4, die die absteigenden Trends für die Javadoc Code Smells darstellt, ist es interessant zu sehen, dass es mehr Projekte gibt, bei denen die fehlenden Tag-Bedingungen und unzulässigen Tags gleichbleibend oder schwankend sind, als die, die einen absteigenden Trend für diese Smell-Kategorien besitzen. Damit ist gemeint, dass die Projekte mit einem absteigenden Javadoc Code Smell Trend häufiger keinen Trend für die fehlenden Tag-Bedingungen und unzulässigen Tags besitzen, als einen absteigenden Trend.

Code Smells / absteigender Trend	short Description	no Javadoc	missing Tag	missing Tag-Requirement	unallowed Tag	missing Package-info
Trend aufsteigend	1	2	2	1	0	0
Trend absteigend	3	4	4	2	1	4
kein Trend	2	0	0	3	5	2

Tabelle 4.4.: Übersicht der Mann-Kendall-Trendanalyse aus der Sicht der Code Smells auf die Projekte mit absteigendem Code Smell Trend.

Abschließend lässt sich über den Verlauf der Javadoc Code Smells der analysierten Projekten sagen, dass sich die Anzahl der Javadocables proportional zu der Anzahl der Javadoc Code Smells verhält.
Ebenso verhalten sich die Javadoc Code Smells zu allen Oberkategorien größtenteils proportional. Des Weiteren steigen in den meisten Fällen die Javadoc Code Smells im Verlauf eines Projektes. Dieses Ergebnis der Analyse unterstreicht die Aussage, dass die Dokumentationen der Systeme vernachlässigt werden.

4.2.2. RQ2 - Häufigkeit der Javadoc Verstöße in den Projekten

Mittels dieser Forschungsfrage wird die Verteilung der Javadoc Code Smells auf die verschiedenen Javadocables sowie die Häufigkeit des Auftretens dieser Code Smells untersucht.
Als Erstes wird die Häufigkeit des Auftretens der verschiedenen Javadoc Code Smells analysiert. Dazu bietet die Abbildung 4.1 einen Überblick über alle 753509 gefundenen Javadoc Code Smells der analysierten Projekte. In dieser Abbildung wurde die Zusammenfassung der 35 Code Smells in die sechs Oberkategorien der ersten Forschungsfrage verwendet, da die Darstellung ansonsten zu unübersichtlich geworden wäre. Fünf der sechs Oberkategorien werden darauf folgend einzeln aufgeschlüsselt. Lediglich die Kategorie der unzulässigen Tags erhält kein eigenes Diagramm, da diese Kategorie nur aus einem Code Smell besteht. Es werden nur 35 der 37 definierten Code Smells analysiert, da die zwei Code Smells für eine zu kurze Short- oder Long-Description nicht in der Konfiguration konfiguriert wurden. Dies ist nicht nötig, da die Total-Description untersucht wird.

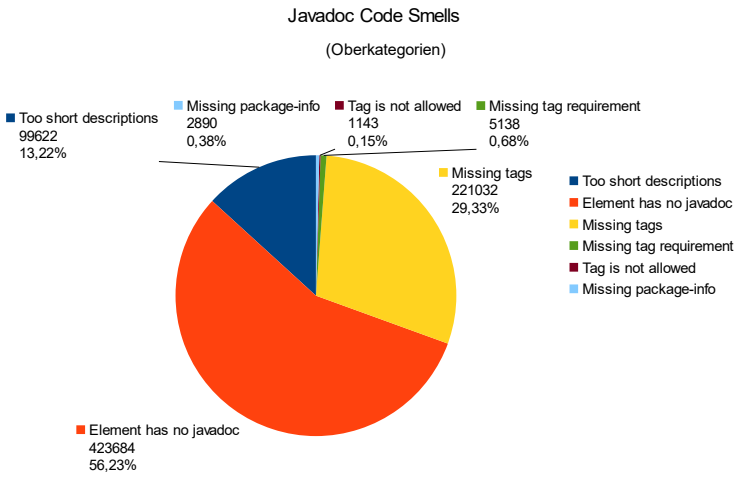


Abbildung 4.1.: Übersicht der gefundenen Javadoc Code Smells in den analysierten Projekten.

Wie in der Forschungsfrage vermutet wurde, ist die Oberkategorie der *"no Javadoc"* - Smells am häufigsten vertreten. Sie nimmt dabei über die Hälfte aller Javadoc Code Smells ein, das deutet, wie in der Formulierung der Forschungsfrage beschrieben, darauf hin dass Dokumentationen vernachlässigt werden. Des Weiteren sind circa ein Drittel der Code Smells auf die Kategorie *"missing Tag"* zurückzuführen. Die Oberkategorie für zu kurze Beschreibungen nimmt dabei circa 13,22% ein, wobei die, für die Analyse, festgelegten Mindestlängen großzügig bemessen wurden. Die festgelegten Mindestlängen können aus der Konfiguration, die für die Analyse verwendet wurde, entnommen werden. Die verwendete Konfiguration wird im Kapitel "Konfigurationsmöglichkeiten" als Konfigurationsbeispiel dargestellt. Die restlichen drei Oberkategorien sind, zusammengefasst mit 1,21%, gering vertreten.

In der nachfolgenden Abbildung 4.2 wird die Verteilung der *"no Javadoc"* - Smells dargestellt.

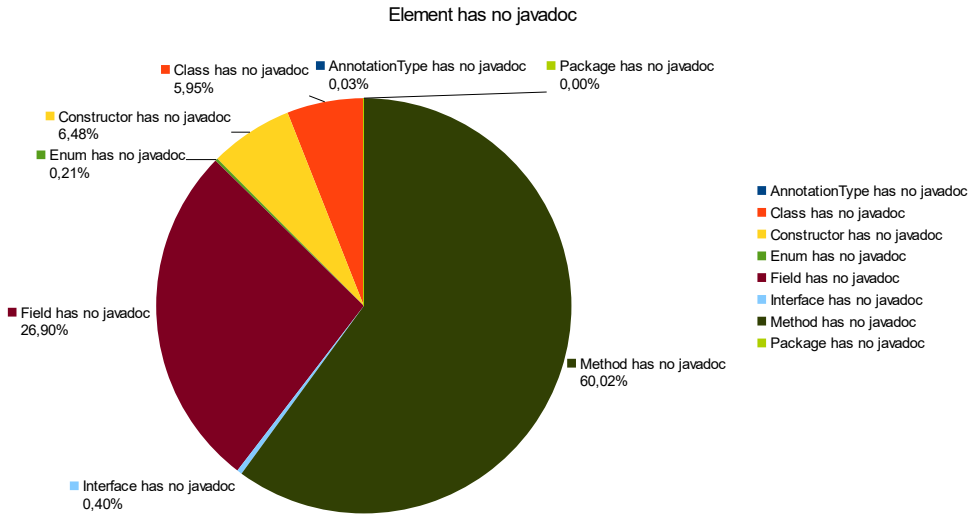


Abbildung 4.2.: Übersicht der Verteilung der *"no Javadoc"* - Smells.

Hierbei ist deutlich zu sehen, dass die Methoden am häufigsten von dem Smell betroffen sind, am zweithäufigsten die Felder. Dies kann unter anderem darauf zurückgeführt werden, dass einige Javadocables viel häufiger in den Projekten vorkommen als andere, dies wird in der dritten Forschungsfrage näher betrachtet. Bei den Paketen ist allerdings das Problem, dass dieser Smell nur gefunden werden kann, sofern das Paket eine Package-Info-Datei ohne Javadoc-Kommentar besitzt. Deshalb müssten eigentlich noch die Pakete dazugezählt werden, die über die Konfigurationen *"Paket benötigt Package-info"* und *"typenloses Paket benötigt Package-info"* gefunden werden, da diese Pakete ohne Package-Info-Datei auch keinen Javadoc-Kommentar besitzen können. Allerdings geht es in dieser Kategorie nur, um die gefundenen *"no Javadoc"* - Smells, weshalb die eben genannten Fälle in der Auswertung ignoriert werden können.
Es wurden 254300 Mal der *"no Javadoc"* - Smell für Methoden, 113964 Mal für Felder, 27456 Mal für Konstruktoren, 25223 Mal für Klassen, 1710 Mal für Interfaces, 882 Mal für Enums, 145 Mal für Annotations und 4 Mal für Pakete gefunden. Wenn allerdings die zwei Konfigurationen berücksichtigt worden wären, wären 2894 Pakete ohne Javadoc-Kommentare gefunden worden. Dadurch wären diese öfters vertreten als die Interfaces.

Im Folgenden werden die Häufigkeiten nicht explizit durch eine Zahl genannt, sondern lediglich durch ihre prozentuale Verteilung. Die genauen Zahlen für diese Verteilung können bei Bedarf aus der Tabelle 4.5 entnommen werden.

In der nachfolgenden Abbildung 4.3 wird die Verteilung der *"missing Tag"* - Smells dargestellt.

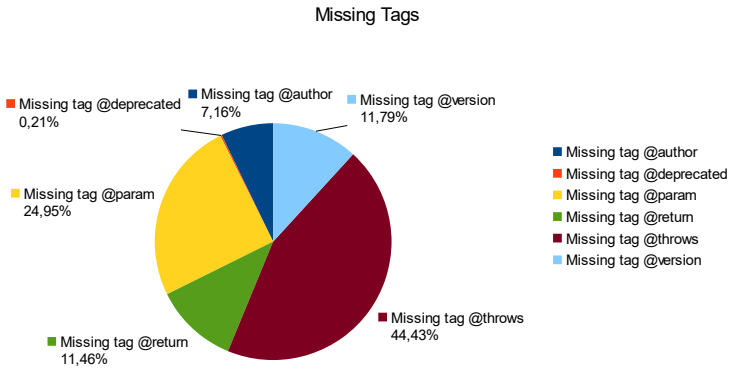


Abbildung 4.3.: Übersicht der Verteilung der *"missing Tag"* - Smells.

Im Vergleich zu dem vorherigen Kreisdiagramm der *"no Javadoc"* - Smells ist dieses Diagramm deutlich gleichmäßiger verteilt. Dabei sticht jedoch der Code Smell *"missing tag @throws"* als Vorreiter heraus, jedoch muss beachtet werden, dass dieser Smell auch den Javadoc-Tag *"@execption"* beinhaltet. Mit fast 20% Unterschied ist der am zweithäufigsten vertretende Smell, der für die fehlende Parameterbeschreibungen. Diese 20% Unterschied lassen sich dadurch erklären, dass die Entwickler häufiger vergessen rekursiv geworfene Exceptions, als die Parameter zu beschreiben. Ein Grund dafür könnte sein, dass die automatische Javadoc-Generierungshilfe der IDEs, die rekrusiv geworfenen Exceptions, bei der Erstellung eines Kommentars nicht berücksichtigen. Für die Parameter, den Return-Wert sowie den lokalen Exceptions eines Executables werden mittels dieser automatischen Javadoc-Generierungshilfe beim Erstellen des Javadoc-Kommentars die dazugehörigen Javadoc-Tags bereits mit angelegt. Die Smells *"missing tag @return"* sowie *"missing tag @author"* und *"missing tag @version"* halten sich dabei im Gleichgewicht. Lediglich der Code Smell für eine nicht dokumentierte Deprecated-Annotation ist kaum vertreten.

In der nachfolgenden Abbildung 4.4 wird die Verteilung der *"short Description"* - Smells dargestellt.

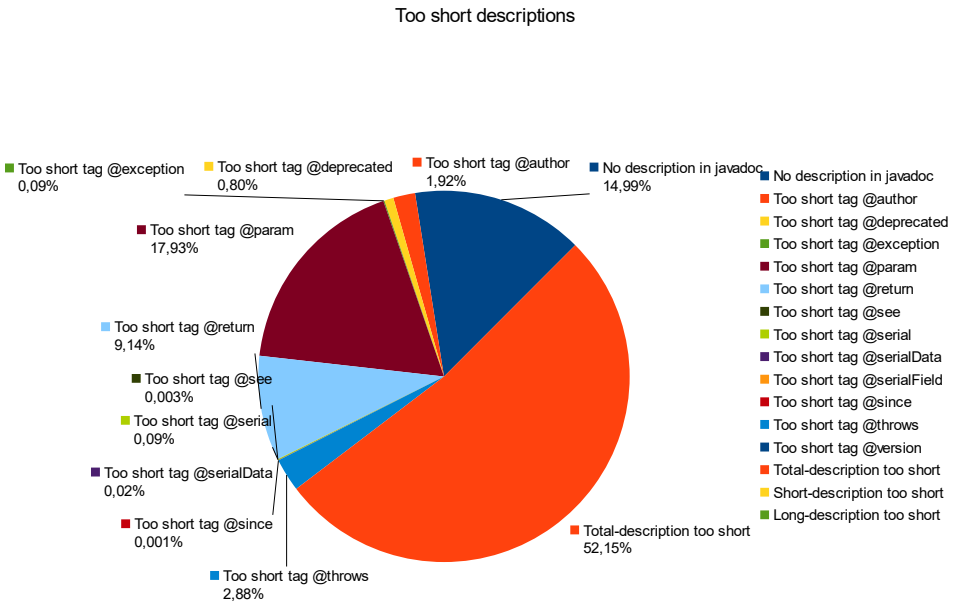


Abbildung 4.4.: Übersicht der Verteilung der *"too short Description"* - Smells.

Hierbei ist gut zu erkennen, dass die Beschreibung eines Javadoc-Kommentars häufiger die vorgegebene Mindestlänge nicht erreicht, als die der Javadoc-Tags. Denn der am häufigsten gefundene Smell in dieser Oberkategorie ist der, der für eine zu kurze Beschreibung der Javadoc-Beschreibung erkannt wird. Dieser bildet zusammen mit dem drittplatzierten Smell *"No description in javadco"* die ursprüngliche Unterkategorie *"too short Description"*. Diese Unterkategorie nimmt 67,14% der gesamten zu kurzen Beschreibungen ein.
Dahingegen nimmt die ursprüngliche Unterkategorie *"to short Tag-Description"* die verbleibenden 32,86% ein. Mit knapp 35% Unterschied zum ersten Platz folgt auf dem zweiten Platz die zu kurze Parameterbeschreibung. Die Smells die jeweils über 1% der Häufigkeiten an sich nehmen, sind die Smells für eine zu kurze Beschreibung des Return-Wertes, der Exception und die des zu kurzen Autors. Dabei ist der Smell für den Autor verwunderlich, da ein Name immer aus einem Vor- und Nachnamen besteht. Wenn der vollständige Vorname, aus irgendwelchen Gründen, nicht in die Dokumentation geschrieben werden kann, dann sollte dieser wenigstens durch den ersten Buchstaben und einem Punkt abgekürzt werden. Außerdem ist interessant, dass in der Analyse kein einziges Mal ein Code Smell für eine zu kurze Versionsbeschreibung oder einer zu kurzen Serial-Fields Beschreibung gefunden wurde. Die Smells für eine zu kurze Short- oder Long-Description entfallen hierbei, wie am Anfang

dieses Unterkapitels beschrieben wurde.

In der nachfolgenden Abbildung 4.5 wird die Verteilung der *"missing Tag-Requirement"* - Smells dargestellt.

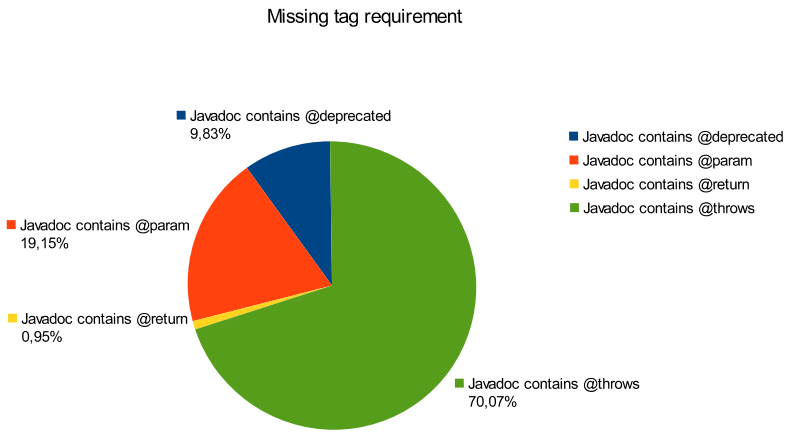


Abbildung 4.5.: Übersicht der Verteilung der *"missing tag requirement"* - Smells.

In der Oberkategorie *"missing Tag-Requirement"* kommt mit ca. 70% am häufigsten der Code Smell vor, der auftritt, wenn eine Exception beschrieben wurde, die von dem Executable nicht geworfen werden kann. Der große Unterschied von fast 51% zum zweitplatzierten Smell kann durch verschiedene Ansätze begründet werden. Zum einen gibt es die Möglichkeit, dass der Executable, der den Smell *"Javadoc contains @throws.."* aufweisen kann, immer noch eine Exception beschreibt die er aufgrund von Ablaufänderungen nicht mehr werfen kann. Dies ist zum Beispiel dann der Fall, wenn der Executable in einer früheren Version einen anderen Executable aufruft, der diese Exeception werfen kann, aber in der analysierten Version diesen Aufruf nicht mehr verwendet und die überflüssige Dokumentation der rekursiven Exception nicht entfernt wurde. Zum anderen besteht die Möglichkeit, dass eine Exception in dem Javadoc-Kommentar beschrieben wurde, die aufgrund einer Behandlung mittels eines *"try-catch"* - Blockes nicht mehr explizit geworfen werden kann.
Auf dem zweiten Platz ist der Smell für eine Parameterbeschreibung eines nicht existierenden Parameters zu finden.
Den dritten Platz nimmt der Code Smell ein, der für eine fehlende Deprecated-Annotation an der Deklaration des jeweiligen Javadocables steht. Als Letztes sind die Code Smells für einen beschriebenen Return-Wert zu finden, obwohl die Methode nichts zurückgibt. Diese nehmen jedoch nicht einmal 1% in der Verteilung ein.

In der nachfolgenden Abbildung 4.6 wird die Verteilung der *"missing Package-info"* - Smells dargestellt.

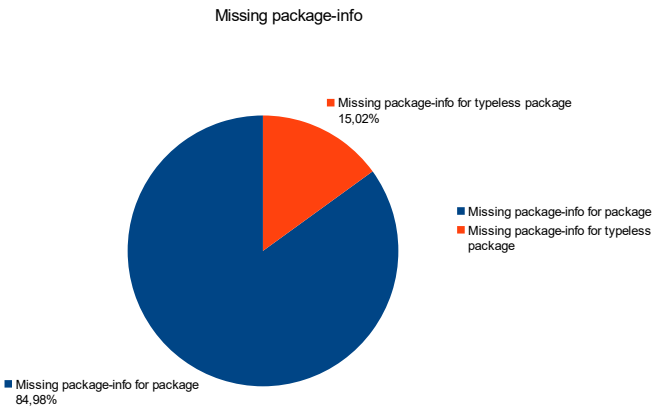


Abbildung 4.6.: Übersicht der Verteilung der *"missing package-info"* - Smells.

Hierbei ist deutlich zu erkennen, dass der Smell *"missing package-info for package"* viel häufiger vorkommt als der Smell *"missing package-info for typeless package"*. Dies könnte darauf zurückzuführen sein, dass es viel mehr Pakete gibt, die einen Typen deklarieren, als typenlose Pakete. Anderenfalls wären typenlose Pakete einfach besser dokumentiert als normale Pakete.

Als Nächstes wird die Verteilung dieser eben genannten Häufigkeiten auf die verschiedenen Javadocables untersucht. Dabei bietet die Abbildung 4.7 einen Überblick über die prozentuale Verteilung der Smells auf die jeweiligen Javadocables. Dabei ist natürlich, wie eben schon erwähnt, zu beachten, dass einige Javadocables viel Häufiger in den Projekten vorkommen als andere. Dieses Problem wird in der dritten Forschungsfrage behandelt.

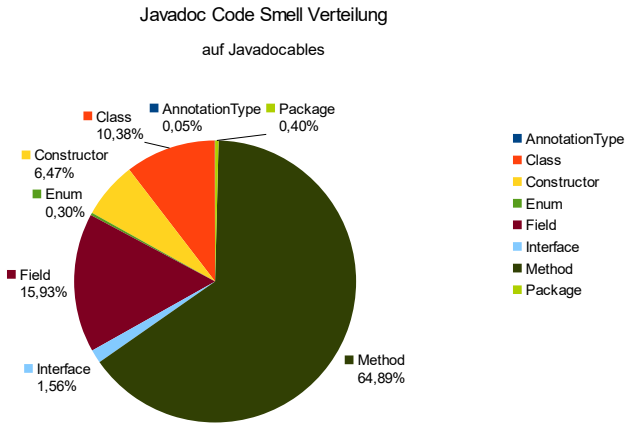


Abbildung 4.7.: Verteilung der gefundenen Javadoc Code Smells auf die Javadocables.

Das eben beschriebene Problem ist in dieser Abbildung deutlich zu erkennen, 65% aller gefundenen Javadoc Code Smells können dem Javadocable der Methoden zugewiesen werden. Auf dem zweiten Platz mit fast 50% Abstand sind die Code Smells der Felder zu finden. Der dritte Platz liegt dabei knapp noch über der zehn Prozent, dabei handelt es sich um die Code Smells der Klassen. Die restlichen circa zehn Prozent der Code Smells werden auf die verbliebenen Javadocables aufgeteilt. Dabei werden den Javadocables Annotationstypen, Pakete und Enums zusammen nicht einmal ein Prozent aller gefundenen Javadoc Code Smells zugeordnet.
Abschließend werden im Folgenden die Javadoc Code Smell Verteilung der einzelnen Javadocables analysiert.
In der Abbildung 4.8 wird die Javadoc Code Smell Verteilung auf die Annotationstypen gezeigt. Am häufigsten sind in diesem Javadocable die Code Smells für keinen Javadoc-Kommentar vorhanden vertreten. Dicht gefolgt von den Smells für eine zu kurze gesamte Beschreibung des Kommentars. Mit zehn Prozent fallen die Smells für einen unzulässigen Javadoc-Tag auf dem dritten Platz. Danach folgen die Smells für eine zu kurze Beschreibung des Autoren-Tags und der Smell, dass keine Beschreibung in dem Javadoc-Kommentar vorhanden ist, auf Platz vier und fünf.

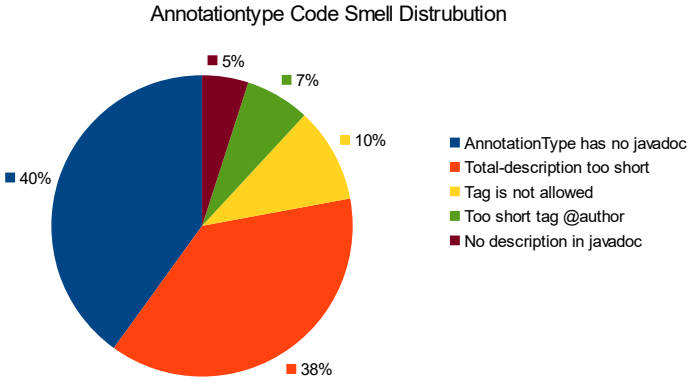


Abbildung 4.8.: Verteilung der gefundenen Javadoc Code Smells für die Annotationstypen.

Mit der Hilfe des Kreisdiagramms in der Abbildung 4.9 wird die Verteilung der Javadoc Code Smells auf die Klassen gezeigt. Dabei ist zu beachten, dass die Legende auf der rechten Seite absteigend geordnet ist. So ist zum Beispiel an der dritten Position der Smell für den vergessenen Tag *"@author"*, diese Smells nehmen im Kreisdiagramm die drittgrößten Verteilung mit 16,778% ein. Wie schon bei den Annotationstypen ist auch bei den Klassen der Smell für den fehlenden Javadoc-Kommentar am häufigsten vertreten. Danach folgt der Code Smell für die fehlende Versionsbeschreibung und anschließend die eben genannte fehlende Autorenbeschreibung. Erst an der vierten Stelle sind hier die Smells für eine zu kurze Beschreibung zu finden. An der fünften Stelle sind die Smells für einen nicht beschriebenen generischen Parameter zu finden und an sechster Stelle, dass keine Beschreibung in dem Javadoc-Kommentar vorhanden ist. Insgesamt wurden 15 verschiedene Javadoc Code Smell - Arten für Klassen gefunden.

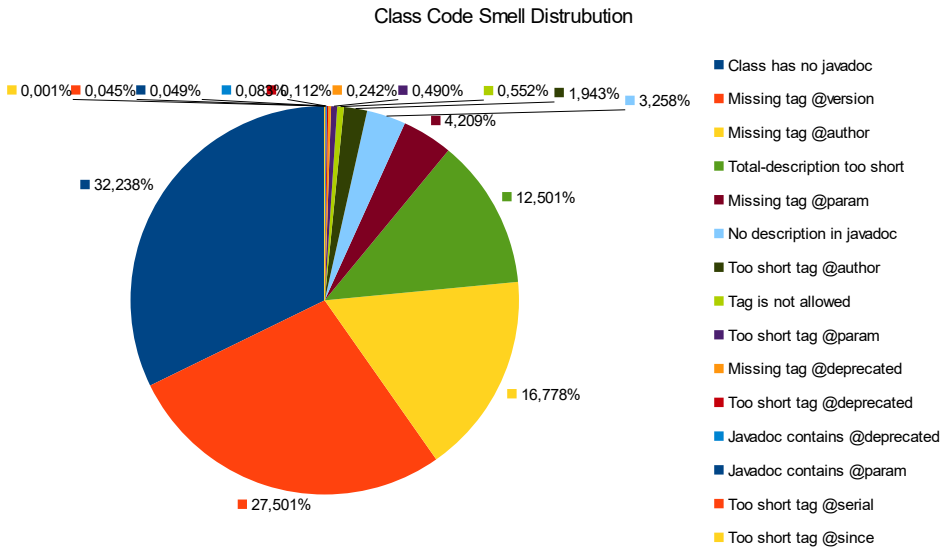


Abbildung 4.9.: Verteilung der gefundenen Javadoc Code Smells für die Klassen.

Die Abbildung 4.10 zeigt die Verteilung der Javadoc Code Smells auf Konstruktoren. Hierbei wurden insgesamt 14 verschiedene Smell - Arten für die Konstruktoren gefunden. Über die Hälfte der gefundenen Smells sind dabei wieder die Smells für keinen vorhandenen Javadoc-Kommentar. Danach folgt auf Platz zwei der meist vertretenden Smells der Klassen, mit einem Unterschied von fast 39%, die Smells für eine fehlende Exceptionbeschreibung. Auf dem dritten Platz sind die Smells für eine zu kurze Beschreibung der Konstruktoren zu finden. Die restlichen Smells sind jeweils unter zehn Prozent vertreten. Dabei stechen jedoch noch die Code Smells für eine vergessene Parameterbeschreibung und die Smells für eine zu kurze Parameterbeschreibung heraus.

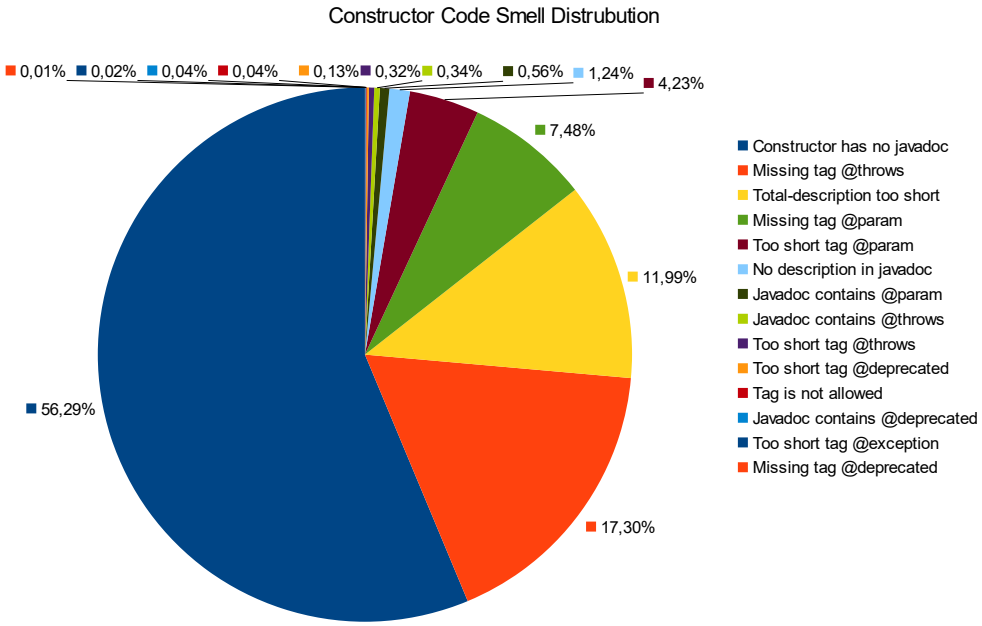


Abbildung 4.10.: Verteilung der gefundenen Javadoc Code Smells für die Konstruktoren.

In der nachfolgenden Abbildung 4.11 wird die Verteilung der neun verschiedenen Javadoc Code Smell Arten auf die Enums dargestellt. Wie schon bei den vorherigen Javadocables, sind auch hier die Smells die keinen vorhandenen Javadoc-Kommentar repräsentieren am häufigsten vertreten. Platz zwei und drei liegen nah beieinander, diese trennen lediglich circa drei Prozent. Auf Platz zwei sind die fehlenden Versionsbeschreibungen zu finden, dies betrifft jedoch nur Enums, die von keinen anderen Typen deklariert wurden, also alleinstehende Enums. Der dritte Platz der am häufigsten auftretenden Smells der Enums belegt der Smell für eine zu kurze Beschreibung. Circa 12,5% der gefundenen Enum Javadoc Code Smells sind auf fehlende Autorenbeschreibungen zurückzuführen. Der Smell, dass die Enums keine Beschreibung in ihrem Javadoc-Kommentar enthalten, ist bei cira drei Prozent der gefundenen Enum Javadoc Code Smells der Fall. Die verbleibenden vier Code Smells bilden zusammengefasst gerade einmal ein wenig mehr als zwei Prozent der gefundenen Javadoc Code Smells in den Enums.

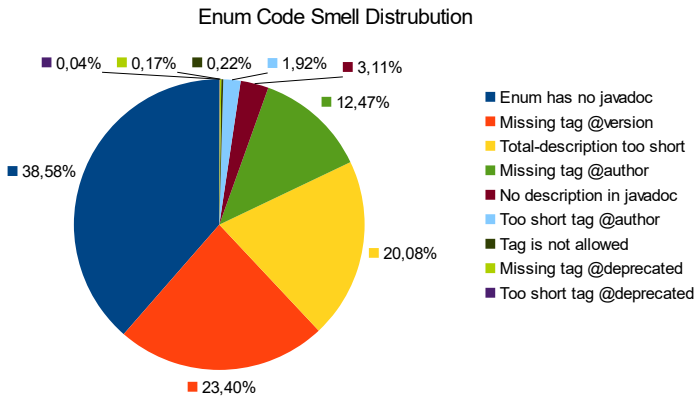


Abbildung 4.11.: Verteilung der gefundenen Javadoc Code Smells für die Enums.

Die Verteilung der Javadoc Code Smells auf die Felder, wie sie in der Abbildung 4.12 zu sehen ist, wird sehr stark von dem Smell für keinen vorhandenen Javadoc-Kommentar geprägt. Dabei nimmt dieser Code Smell alleine schon fast 95% aller gefundenen Smells für Felder ein. An dieser Stelle ist fraglich, ob es für die Analyse sinnvoller gewesen wäre nur die öffentlichen Felder, also Felder mit dem Zugriffsmodifikator *"public"*, zu betrachten, da private Felder oftmals nicht dokumentiert werden. Jedoch nehmen der zweite und dritte Platz noch jeweils über ein Prozent der Verteilung ein. Es handelt sich bei den beiden Smells, um die Smells für eine zu kurze Beschreibung sowie dass keine Beschreibung in dem Javadoc-Kommentar vorhanden ist.

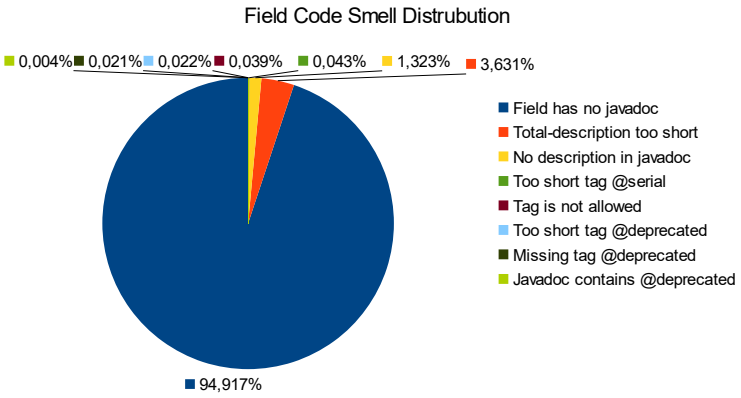


Abbildung 4.12.: Verteilung der gefundenen Javadoc Code Smells für die Felder.

Dahingegen zeigt die Abbildung 4.13, der Verteilung der Javadoc Code Smells auf die Interfaces, eine schönere Verteilung der Smells. Insgesamt wurden hierbei 13 verschiedene Javadoc Code Smell Arten gefunden. Auffällig ist, dass der erste Platz nicht durch die Smells für *"kein Javadoc-Kommentar vorhanden"* belegt wird, sondern von dem Smell für die fehlende Versionsbeschreibung. Am zweithäufigsten ist der Smell für die fehlende Autorenbeschreibung zu finden. Der dritte Platz wird von dem Smell für eine zu kurze Beschreibung belegt und erst der am vierthäufigsten auftretende Smell, ist der für den Fall, dass kein Javadoc-Kommentar vorhanden ist. Mit circa acht Prozent sticht der Smell für eine fehlende Parameterbeschreibung für einen generischen Parameter noch ein wenig von den restlichen Smells heraus. Lediglich 2,55% der gefundenen Smells konnten auf den Smell, für keine Beschreibung in dem Javadoc-Kommentar vorhanden, zurückgeführt werden.

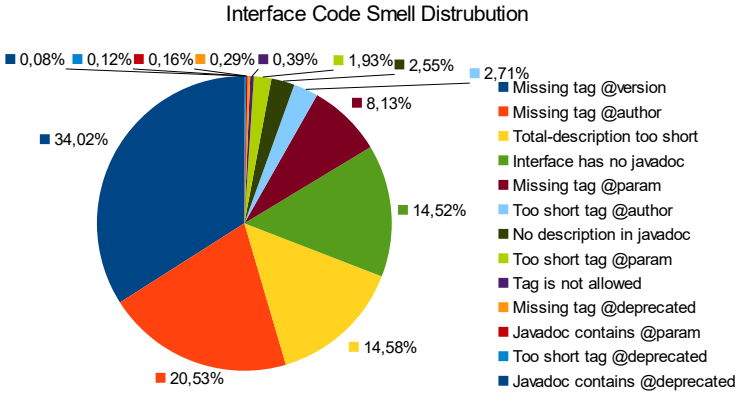


Abbildung 4.13.: Verteilung der gefundenen Javadoc Code Smells für die Interfaces.

Die Abbildung 4.14 stellt die Verteilung der Javadoc Code Smells auf die Methoden dar. Hierbei wurden insgesamt 19 verschiedene Javadoc Code Smell Arten für Methoden gefunden. Am häufigsten wurden dabei erneut die Smells für keinen vorhanden Javadoc-Kommentar erkannt. Den Platz für die zweithäufigsten vertretenden Smells bilden die Smells für eine nicht beschriebene Exception. Die nachfolgenden Smells liegen dabei jeweils unter zehn Prozent. Circa 9,66% der gefundenen Javadoc Code Smells der Methoden waren die Smells für eine fehlende Parameterbeschreibung, diese bilden somit den dritten Platz. Eine zu kurze Beschreibung war in circa sechs Prozent der Smells der Fall. Außerdem wurden in circa fünf Prozent der Smells vergessen der Return-Wert der Methode zu beschreiben. Die restlichen Smells sind dabei relativ wenig vertreten und werden deshalb nicht explizit genannt.

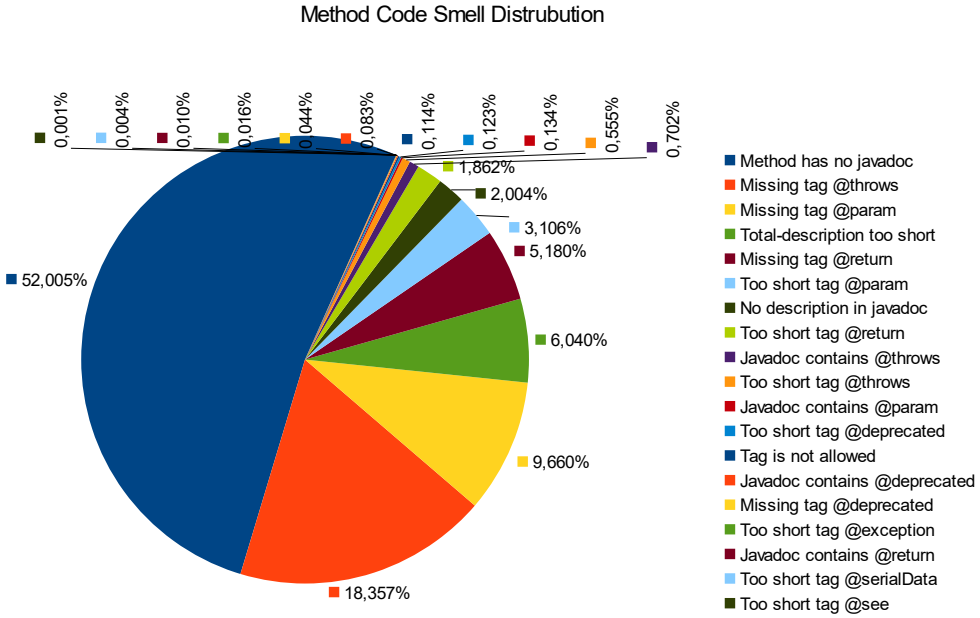


Abbildung 4.14.: Verteilung der gefundenen Javadoc Code Smells für die Methoden.

Abschließend wird in der Abbildung 4.15 die Verteilung der Code Smells auf die Pakete dargestellt. Hierbei ist interessant zu sehen, dass die Smells für eine fehlende Package-Info-Datei für typendeklarierende Pakete am häufigsten vertreten sind und den Großteil der Verteilung ausmachen. Auf dem zweiten Platz befindet sich der Smell für die fehlenden Package-Info-Dateien der typenlosen Pakete sowie auf dem dritten die zu kurze Beschreibung der Pakete. Erst auf dem vierten Platz ist der Code Smell Bereich für kein Javadoc-Kommentar zu finden, dieser macht gerade einmal 0,13% der Smells aus. Jedoch gehören die Code Smells für die fehlenden Package-Info-Dateien und die des "kein Javadoc vorhanden" eigentlich, wie vorhin für die Abbildung 4.2 beschrieben, zusammen.

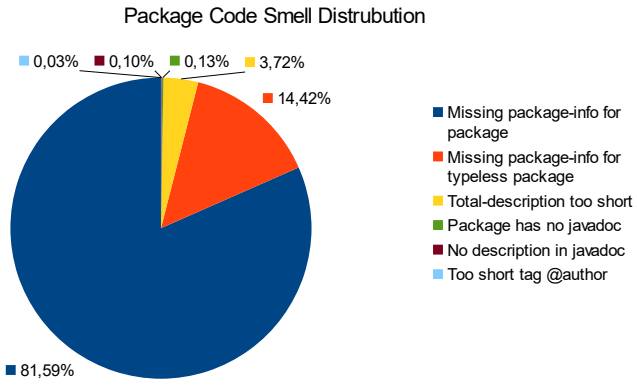


Abbildung 4.15.: Verteilung der gefundenen Javadoc Code Smells für die Pakete.

Dabei ist es nicht verwunderlich, dass die Code Smells für *"keinen Javadoc vorhanden"* am häufigsten in den meisten Javadocables auftritt, denn wie bereits in den Grundlagen beschrieben wurde, wird die Dokumentation der Systeme oft vernachlässigt. Erstaunlich hingegen ist, dass diese Smells für Interfaces am wenigsten aufgetreten sind, daraus folgt, dass an einem Interface am ehesten ein Javadoc-Kommentar zu finden ist.
Die Daten für die Kreisdiagramme wurden aus der nachfolgenden Tabelle 4.5 entnommen.

Javadocables / Javadoc Code Smells	AnnotationType	Class	Constructor	Enum	Field	Interface	Method	Package	$\sum_{i=0}^{n}$ (Anzahl Tag)
AnnotationType has no javadoc	145	0	0	0	0	0	0	0	145
Class has no javadoc	0	25223	0	0	0	0	0	0	25223
Constructor has no javadoc	0	0	27456	0	0	0	0	0	27456
Enum has no javadoc	0	0	0	882	0	0	0	0	882
Field has no javadoc	0	0	0	0	113964	0	0	0	113964
Interface has no javadoc	0	0	0	0	0	1710	0	0	1710
Javadoc contains @deprecated	0	65	18	0	5	9	408	0	505
Javadoc contains @param	0	38	274	0	0	19	653	0	984
Javadoc contains @return	0	0	0	0	0	0	49	0	49
Javadoc contains @throws	0	0	165	0	0	0	3435	0	3600
Method has no javadoc	0	0	0	0	0	0	254300	0	254300
Missing package-info for package	0	0	0	0	0	0	0	2456	2456
Missing package-info for typeless package	0	0	0	0	0	0	0	434	434
Missing tag @author	0	13127	0	285	0	2419	0	0	15831
Missing tag @deprecated	0	189	7	4	25	34	213	0	472
Missing tag @param	0	3293	3650	0	0	958	47236	0	55137
Missing tag @return	0	0	0	0	0	0	25329	0	25329
Missing tag @throws	0	0	8440	0	0	0	89763	0	98203
Missing tag @version	0	21517	0	535	0	4008	0	0	26060
No description in javadoc	18	2549	607	71	1588	300	9797	3	14933
Package has no javadoc	0	0	0	0	0	0	0	4	4
Tag is not allowed	37	432	19	5	47	46	557	0	1143
Too short tag @author	25	1520	0	44	0	319	0	1	1909
Too short tag @deprecated	0	88	63	1	26	14	603	0	795
Too short tag @exception	0	0	11	0	0	0	80	0	91
Too short tag @param	0	383	2062	0	0	227	15186	0	17858
Too short tag @return	0	0	0	0	0	0	9104	0	9104
Too short tag @see	0	0	0	0	0	0	3	0	3
Too short tag @serial	0	35	0	0	52	0	0	0	87
Too short tag @serialData	0	0	0	0	0	0	20	0	20
Too short tag @serialField	0	0	0	0	0	0	0	0	0
Too short tag @since	0	1	0	0	0	0	0	0	1
Too short tag @throws	0	0	155	0	0	0	2716	0	2871
Too short tag @version	0	0	0	0	0	0	0	0	0
Total-description too short	137	9781	5849	459	4360	1717	29535	112	51950
Short-description too short	0	0	0	0	0	0	0	0	0
Long-description too short	0	0	0	0	0	0	0	0	0
$\sum_{i=0}^{n}$ **(Javadoc Code Smells)**	362	78241	48776	2286	120067	11780	488987	3010	753509

Tabelle 4.5.: Übersicht über die gefundenen Javadoc Code Smells in den analysierten Projekten.

4.2.3. RQ3 - Von Javadoc Code Smells befallene oder freie Javadocables

Mit der Hilfe dieser Forschungsfrage wird untersucht, zu wie viel Prozent ein Javadocable von Javadoc Code Smells befallen oder eben nicht befallen ist.
Das nachfolgende Kreisdiagramm der Abbildung 4.16 stellt einen Überblick darüber dar, wie häufig die verschiedenen Javadocables in den analysierten Projekten vorkommen. Dabei ist deutlich zu erkennen, dass folgende absteigende Rangordnung für die Javadocables der analysierten Projekte existiert:
Anzahl Methoden > Anzahl Felder > Anzahl Konstruktoren > Anzahl Klassen > Anzahl Interfaces > Anzahl Pakete > Anzahl Enums > Anzahl Annotationstypen

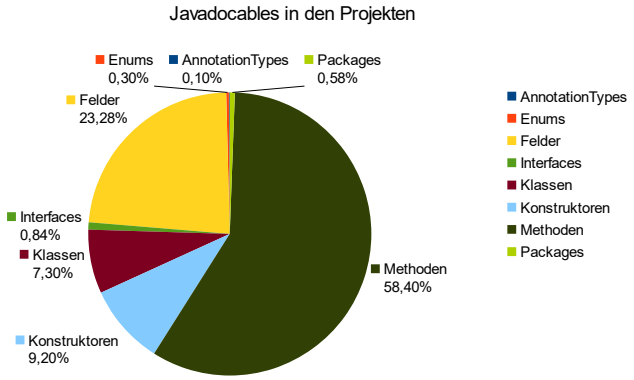


Abbildung 4.16.: Übersicht über die Häufigkeiten der einzelnen Javadocables aus den analysierten Projekten.

Im Nachfolgenden wird untersucht, wie viele von den eben genannten Javadocables von Javadoc Code Smells befallen und wie viele frei von diesen Smells sind. Ein Javadocable gilt als befallen, sobald dessen Javadoc-Kommentar einen Javadoc Code Smell enthält, die genaue Anzahl der Smells in dem Kommentar spielt dabei jedoch keine Rolle.
Dazu bietet das folgende Kreisdiagramm in Abbildung 4.17 eine Übersicht über die Verteilung der befallenen Javadocables, jedoch unter Berücksichtigung ihrer Häufigkeiten. Dabei zeigt sich im Kreisdiagramm eine geänderte Rangordnung, die sich aus der Änderung der Rangordnung zwischen den Klassen und den Konstruktoren ergibt.
Anzahl Methoden > Anzahl Felder > Anzahl Klassen > Anzahl Konstruktoren > Anzahl Interfaces > Anzahl Pakete > Anzahl Enums > Anzahl Annotationstypen
Das bedeutet, dass es mehr von Javadoc Code Smells befallene Klassen als Konstruktoren gibt.

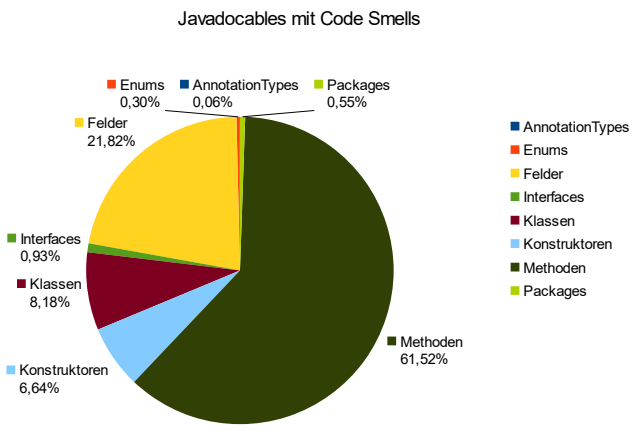


Abbildung 4.17.: Übersicht über die Häufigkeiten der von Javadoc Code Smell befallenden Javadocables.

Jedoch liefert das eben gezeigte Diagramm keine Übersicht darüber, wie die Verteilung unter den Javadocables aussehen würde, wenn die verwendeten Daten normiert wären. Damit ist gemeint, dass die Anzahl der vorkommenden Javadocables auf die Übersicht keine Rolle spielen sollte, sondern lediglich der prozentuale Anteil, ob der jeweilige Javadocable befallen oder unbefallen ist. Dahingegen zeigt das Diagramm in Abbildung 4.18 die Werte der Javadocables, wie eben beschrieben, normiert. Dafür wurde die Anzahl des befallenen Javadocable durch die gesamte Anzahl des vorkommenden Javadocables geteilt und mit den Wert 100 multipliziert, um einen Prozentsatz des Befalls zu erhalten. Dadurch wird im Diagramm ersichtlich, dass in der Regel fast alle Javadocables gleich stark befallen sind. Mit der Ausnahme der Annotationstypen, die nur circa halb so oft befallen sind, wie die anderen Javadocables. Hierfür lässt sich die folgende absteigende Rangordnung bilden, dabei ist der erste Javadocable derjenige, der am häufigsten einen Befall vorweisen kann. Der letzte in der Reihenfolge ist der Javadocable der am wenigsten einen Befall vorweisen kann.
Klassen > Interfaces > Methoden > Enums > Pakete > Felder > Konstruktoren > Annotationstypen
Auf die genauen Prozentzahlen, wie häufig die einzelnen Javadcoables befallen oder unbefallen sind, wird später in diesem Kapitel eingegangen.

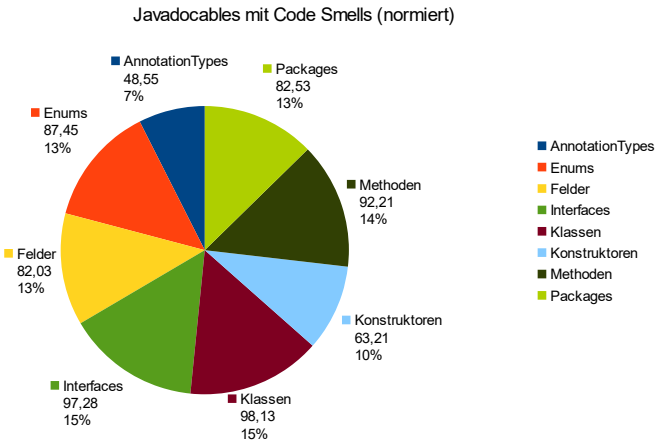


Abbildung 4.18.: Übersicht über die Häufigkeiten der von Javadoc Code Smell befallenden Javadocables (normiert).

Analog wird nun dasselbe auch für die unbefallenen Javadocables in den Diagrammen der Abbildung 4.19 und der Abbildung 4.20 gezeigt. Jedoch setzen sich hierbei die Prozente für das Diagramm der Abbildung 4.20 so zusammen, dass die Anzahl des unbefallenen Javadocable durch die Anzahl des gesamten Vorkommens des Javadocables geteilt und anschließend mit 100 multipliziert wird, um den Prozentsatz zu ermitteln.
Dabei zeigt das zuerst genannte Diagramm die folgende Reihenfolge:
Anzahl Methoden > Anzahl Felder > Anzahl Konstruktoren > Anzahl Klassen > Anzahl Pakete > Anzahl Annotationstypen > Anzahl Enums > Anzahl Interfaces
Demnach sind unter Berücksichtigung der Anzahl der vorkommenden Javadocables Methoden am häufigsten und Interfaces am wenigsten unbefallen.

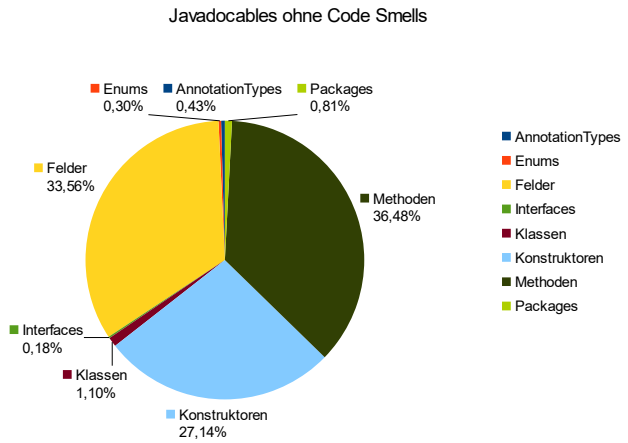


Abbildung 4.19.: Übersicht über die Häufigkeiten der von Javadoc Code Smell unbefallenden Javadocables.

Da die Häufigkeiten der Vorkommnisse der einzelnen Javadocables der Projekte aber diese Verteilung verfälschen, wird in dem zweiten Diagramm diese Verteilung erneut mit normierten Werten betrachtet. Daraus resultiert folgende Reihenfolge:
Annotationstypen > Konstruktoren > Felder > Pakete > Enums > Methoden > Interfaces > Klassen
Prozentual betrachtet sind somit die Annotationtypes am häufigsten frei von Javadoc Code Smells und Klassen am seltensten.

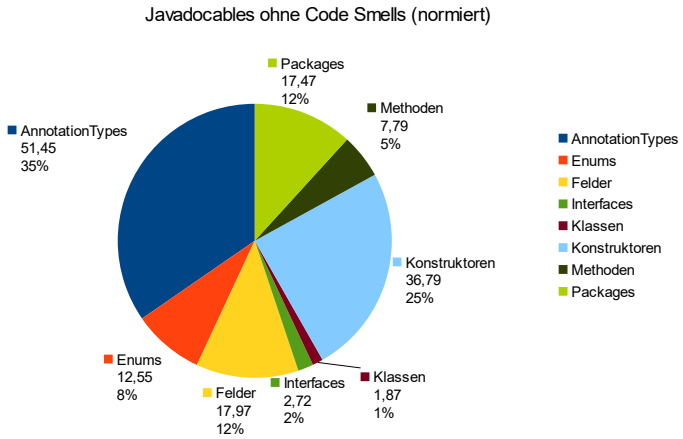


Abbildung 4.20.: Übersicht über die Häufigkeiten der von Javadoc Code Smell unbefallenden Javadocables (normiert).

Im nächsten Schritt wird im Kreisdiagramm der Abbildung 4.21 die prozentuale Verteilung der zusammengefassten befallenen und unbefallen Javadocables gezeigt. Dadurch lässt sich zeigen, dass die Javadocables der analysierten Projekte zu 87,53% von Javadoc Code Smells befallen und nur 12,47% frei von diesen Smells waren.

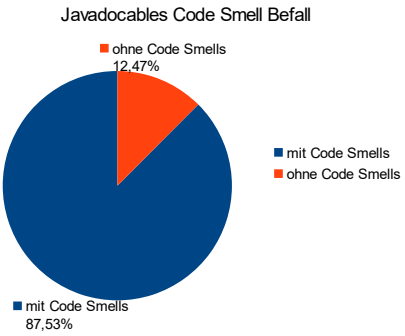
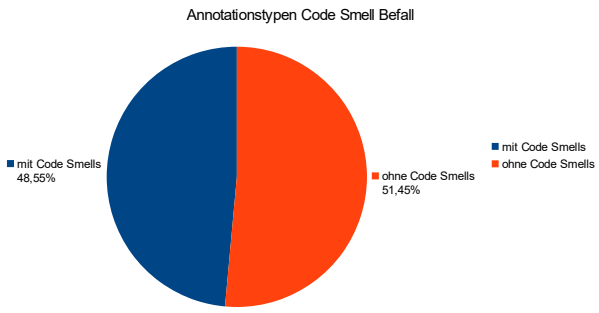


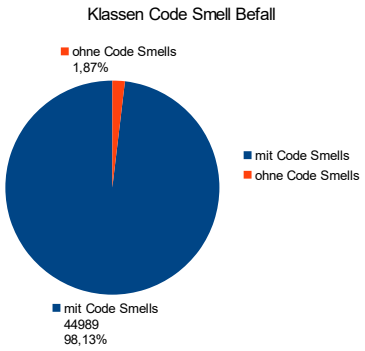
Abbildung 4.21.: Übersicht über die Verteilung zwischen befallene und unbefallene Javadocables.

Abschließend wird nun für jeden einzelnen Javadocable die prozentuale Verteilung zwischen befallen und unbefallen überprüft. Dazu zeigt die Abbildung 4.22 für jeden Javadocable ein eigenes Kreisdiagramm. Daraus geht Folgendes hervor:

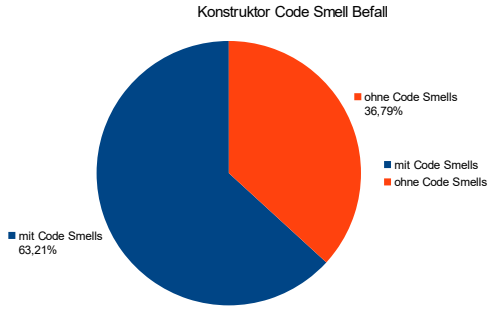
- Annotationtypes sind zu 48,55% von Javadoc Code Smells befallen und zu 51,45% frei von solchen Smells.
- Klassen sind zu 98,13% von Javadoc Code Smells befallen und zu 1,87% frei von solchen Smells.
- Konstruktoren sind zu 63,21% von Javadoc Code Smells befallen und zu 36,79% frei von solchen Smells.
- Enums sind zu 87,45% von Javadoc Code Smells befallen und zu 12,55% frei von solchen Smells.
- Felder sind zu 82,03% von Javadoc Code Smells befallen und zu 17,97% frei von solchen Smells.
- Interfaces sind zu 97,28% von Javadoc Code Smells befallen und zu 2,72% frei von solchen Smells.
- Methoden sind zu 92,21% von Javadoc Code Smells befallen und zu 7,79% frei von solchen Smells.
- Pakete sind zu 82,53% von Javadoc Code Smells befallen und zu 17,47% frei von solchen Smells.



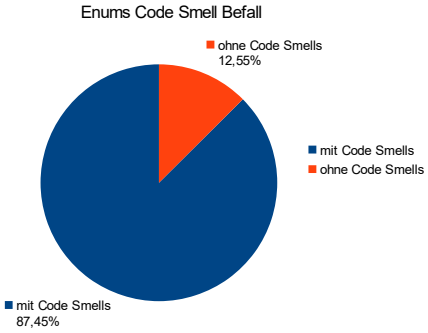
(a) Verteilung der Annotationstypen



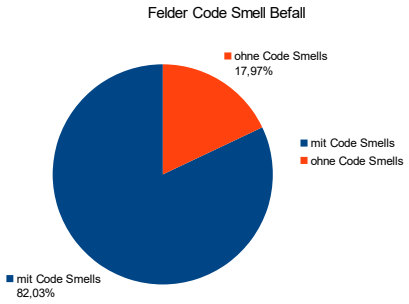
(b) Verteilung der Klassen



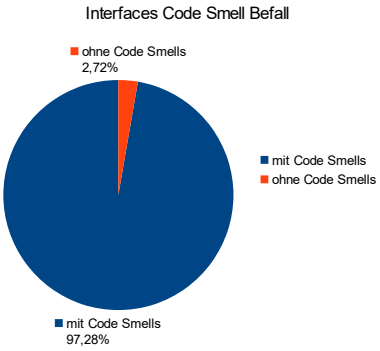
(c) Verteilung der Konstruktoren



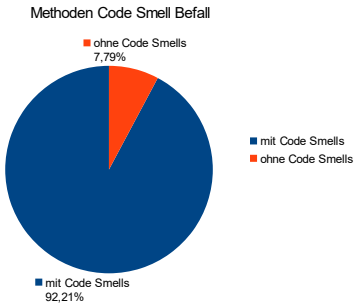
(d) Verteilung der Enums



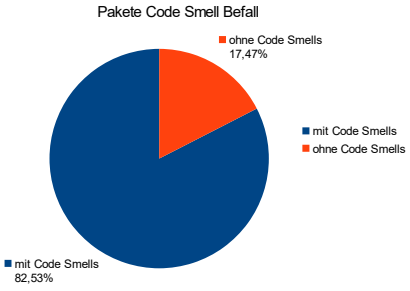
(e) Verteilung der Felder



(f) Verteilung der Interfaces



(g) Verteilung der Methoden



(h) Verteilung der Pakete

Abbildung 4.22.: Die Verteilung zwischen befallenen und unbefallenen für jeden einzelnen Javadocable.

Wie eben gezeigt, sind Klassen somit prozentual betrachtet die am häufigsten von Javadoc Code Smells befallenen und die Annotationstypen die am wenigsten befallenen Javadocables. Es gilt somit die folgende absteigende Reihenfolge für den prozentual betrachteten Befall von Javadoc Code Smells für die Javadocables:

1. Klassen
2. Interfaces
3. Methoden
4. Enums
5. Pakete
6. Felder
7. Konstruktoren
8. Annotationstypen

Die Daten für die Kreisdiagramme aus diesem Unterkapitel wurden aus der nachfolgenden Tabelle 4.6 entnommen.

Anzahl	in Projekten	mit Code Smells	ohne Code Smells	% mit Smells	% ohne Smells
Javadocables	628048	549734	78314	87,53	12,47
AnnotationTypes	657	319	338	48,55	51,45
Enums	1857	1624	233	87,45	12,55
Felder	146240	119956	26284	82,03	17,97
Interfaces	5257	5114	143	97,28	2,72
Klassen	45848	44989	859	98,13	1,87
Konstruktoren	57759	36507	21252	63,21	36,79
Methoden	366784	338216	28568	92,21	7,79
Packages	3646	3009	637	82,53	17,47

Tabelle 4.6.: Übersicht über die Javadocables, die von Javadoc Code Smells befallenen oder frei sind.

5. Schluss

Dieses Kapitel bildet das abschließende Kapitel zu dieser Bachelorarbeit. In den folgenden Unterkapiteln wird ein Fazit zu dieser Arbeit sowie ein Ausblick zu weiteren Forschungsmöglichkeiten und Entwicklungsmöglichkeiten gegeben.

5.1. Fazit

Im Rahmen dieser Bachelorarbeit wurde die LibVCS4j-Bibliothek der Universität Bremen um den umfangreichen Javadoc-Detektor erweitert. Aufgrund dessen, dass es keine expliziten Vorgaben gibt, ab wann ein Javadoc-Kommentar als hinreichend beschrieben gilt, muss dieser Detektor nach den Kriterien des Nutzers konfigurierbar sein.
Dafür bietet der Javadoc-Detektor viele verschiedene Konfigurationsmöglichkeiten. Zum einen kann entschieden werden, welche Javadocables und Zugriffsmodifikatoren in der Analyse berücksichtigt werden sollen. Javadocables bezeichnet dabei die verschiedenen javadocfähigen Elemente, in diesem Detektor umfasst dies die Annotationstypen sowie Enums, Felder, Interfaces, Klassen, Konstruktoren, Methoden und Pakete. Bei den Zugriffsmodifikatoren wird dabei zwischen *“public“* sowie *“private“*, *“protected“* und *“default“* unterschieden.
Des Weiteren lässt sich für jeden Javadocable die Mindestlänge der Javadoc-Beschreibung konfigurieren. Außerdem können ebenfalls verschiedene Mindestlängen für die Javadoc-Tags eingestellt werden. Bei dem Durchlauf der Analyse werden die Beschreibungen und Tags der Javadoc-Kommentare dann hinsichtlich der konfigurierten Mindestlänge untersucht. Einige dieser Javadoc-Tags werden darüber hinaus noch zusätzlich auf andere Eigenschaften überprüft. Dies betrifft die Javadoc-Tags *“@author“*, *“@version“*, *“@deprecated“*, *“@return“*, *“@param“* und *“@throws/@exception“*. Für den Tag des Autors und der Version wird zusätzlich überprüft, ob es sich bei dem untersuchten Javadocable um einen Top-Level-Typen handelt. Ist dies der Fall und ist zusätzlich eine Mindestlänge für diesen Tag konfiguriert worden, wird der Tag in dem dazugehörigen Javadoc-Kommentar erwartet. Bei dem Deprecated-Tag wird zusätzlich geprüft, ob der Javadocable an seiner Deklaration die dazugehörige Deprecated-Annotation besitzt, da die Annotation und der Javadoc-Tag gemeinsam verwendet werden [ORA93a], weshalb auch eine Überprüfung in der anderen Richtung stattfindet. Für den Return-Tag wird überprüft, ob die Methode einen Rückgabewert besitzt und ob dieser beschrieben wurde. Alternativ wird überprüft, ob bei einem beschriebenen Return-Tag die Methode auch tatsächlich etwas zurückgibt. Die zusätzlichen Überprüfungen der Tags für Parameter und Exceptions funktionieren, indem überprüft wird, ob alle existierenden Parameter oder die Exceptions, die geworfen werden können, im Javadoc-Kommentar beschrieben wurden. Außerdem werden bei der Untersuchung der Exceptions auch die Exceptions der rekursiven Aufrufe berücksichtigt. Des Weiteren wird auch die andere Richtung überprüft,

ob alle beschriebenen Exceptions von der analysierten Methode oder Konstruktor geworfen werden können. Außerdem wird überprüft, ob alle Parameter, die im Javadoc-Kommentar beschrieben wurden, auch in diesem Javadocable existieren.
Des Weiteren kann die Weise, wie die Beschreibung gezählt werden soll, gewählt werden. Dabei unterscheidet der Detektor zwischen wortweises und charakterweises zählen. Als Wörter werden dabei Wörter, die durch einen oder mehrere Leerzeichen oder Bindestriche getrennt wurden sowie Zahlen gezählt. Bei dem charakterweisen Zählen werden die Leerzeichen mitgezählt.
Zusätzlich bietet der Detektor die Möglichkeit, die Untersuchung für die Field-Access, also Getter und Setter, zu aktivieren oder zu deaktivieren. Dies wird benötigt, da es Nutzer gibt, die nicht zwingend voraussetzen, dass Getter und Setter dokumentiert sein müssen.
Außerdem bietet der Javadoc-Detektor die Möglichkeit, die Pakete des analysierten Projektes im Hinblick auf die Existenz ihrer Package-Info-Dateien zu überprüfen. Diese Dateien werden benötigt, um eine Dokumentation für ein Paket anzulegen. Ein Paket kann nur durch eine solche Datei dokumentiert werden, an anderer Stelle ist dies nicht zulässig. Mittels einer weiteren Überprüfung wird untersucht, ob die vorhandenen Javadoc-Tags in dem Javadoc-Kommentar überhaupt für den analysierten Javadocable zulässig sind.
Durch diese Konfigurationen ist es dem Javadoc-Detektor möglich, bis zu 37 verschiedene Javadoc Code Smells zu erkennen. Diese 37 verschiedenen Smells können dabei in sieben Oberkategorien kategorisiert werden. Diese Oberkategorien umfassen dabei die folgenden Bereiche der Smells:

- kein Javadoc-Kommentar vorhanden
- fehlende Javadoc-Tags im Javadoc-Kommentar
- fehlende Bedingung, um den dokumentierten Javadoc-Tag zu verwenden
- zu kurze Beschreibungen der Javadoc-Beschreibung
- zu kurze Beschreibung der Javadoc-Tags
- fehlende Package-Info-Dateien der Pakete
- unzulässige Tags

Neben dem beschriebenen Javadoc-Detektor liegen dieser Arbeit zusätzlich drei Forschungsfragen zugrunde.
Die erste dieser Forschungsfragen beschäftigte sich mit dem Thema, wie sich die analysierten Projekte über die Zeit hinweg in Hinblick auf die Javadoc Code Smells entwickeln. Dabei wurde festgestellt, dass die meisten Projekte im Laufe der Zeit immer mehr Javadoc Code Smells aufweisen, was darauf schließen lässt, dass die Dokumentation vernachlässigt wurde.
Die zweite Forschungsfrage widmete sich der Frage, wie sich die gefundenen Javadoc Code Smells auf die verschiedenen Javadocables aufteilen und wie häufig diese auftraten. Diese Fragestellung führte zu dem Ergebnis, dass die Smells der Oberkategorie *"kein Javadoc-Kommentar vorhanden"* am häufigsten auftraten. Erstaunlich hierbei war jedoch, dass Interfaces am wenigsten von dieser Oberkategorie betroffen waren, wenn dieses Auftreten der Smells prozentual betrachtet wird.
Durch die dritte Forschungsfrage wurde untersucht, wie die Verteilung zwischen befallenen und unbefallenen Javadocables der analysierten Projekte ist. Dabei stellte sich heraus, dass die Methoden, wie erwartet, am häufigsten in den Projekten vorkommen. Bei einer prozentualen Betrachtung des Befalls oder nicht Befalls der jeweiligen Javadocables zeigt sich jedoch, dass Klassen am häufigsten und Annotationstypen am seltensten von Javadoc Code Smells betroffen sind.
Mittels dieser drei Forschungsfragen wurde die These von Christian Ullenboom bestätigt, dass die Dokumentation von Systemen in der Softwareentwicklung leider sehr häufig vernachlässigt wird. [Ull12, S. 1259]
Des Weiteren kann die These mittels der drei Forschungsfragen weiter spezifiziert werden. Durch die erste Forschungsfrage ist ersichtlich geworden, dass die Dokumentationen der Projekte im Laufe der Zeit immer mehr Fehler aufwiesen, was bedeutet, dass die ungenügende Dokumentierung der Systeme ein kontinuierlicher Prozess ist. Und somit die Dokumentationen im Laufe der Zeit immer mehr an Qualität verlieren. Mittels der zweiten Forschungsfrage kann die These dahingehend spezifiziert werden, dass der häufigste festgestellte Fehler in den Dokumentationen der ist, dass kein Javadoc-Kommentar für den jeweiligen Javadocable angelegt wurde. Abschließend kann die These durch die dritte Forschungsfrage dahingehend spezifiziert werden, dass Klassen am häufigsten, Interfaces am zweithäufigsten und Methoden am dritthäufigsten von Fehlern in der Dokumentation betroffen sind. Dahingegen sind Annotationstypen am seltensten und Konstruktoren am zweitseltensten von Fehlern in der Dokumentation betroffen.

5.2. Ausblick

In diesem Abschnitt werden abschließend noch einige mögliche Erweiterungen sowie weitere Forschungsmöglichkeiten des Javadoc-Detektors vorgestellt.
Sobald eine bessere Analyse der Javadoc-Tags *"see"* sowie *"serial"*, *"serialData"* und *"serialField"* seitens der Spoon-Bibliothek möglich ist, sollten neben der Längenüberprüfung auch die Inhalte der Tags überprüft werden. Für den Javadoc-Tag *"see"* bedeutet das zum Beispiel, ob die Verweise, auf die der Tag verweist, auch wirklich existieren, sofern es sich dabei um ein Objekt handelt. Bei dem Tag *"serial"* könnte überprüft werden, ob bei einer Klasse oder einem Paket das richtige Schlüsselwort *"include"* oder *"exclude"* verwendet wurde. Des Weiteren könnte überprüft werden, ob das serialisierbare Feld, das mittels des Tags beschrieben wird, existiert. Problematisch ist dabei ist nur, dass die Beschreibung des *"serial"* - Tags laut der Dokumentation von Oracle optional ist. [ORA93b] Für den Tag *"serialData"* könnte überprüft werden, ob es sich bei der Methode, die diesen Tag in ihrem Javadoc-Kommentar beinhaltet, um eine writeObject-, readObject-, writeExternal- , readExternal-, writeReplace- oder readResolve-Methode handelt. Denn die Verwendung dieses Tag ist nur für die eben genannten Methoden erlaubt. Außerdem könnte für den Tag *"serialField"* überprüft werden, ob es sich bei dem untersuchten Feld wirklich um ein ObjectStreamFeld handelt.
Eine zusätzliche Erweiterung für die Javadoc-Tags wäre es, die Inline-Javadoc-Tags zu berücksichtigen und auszuwerten. Wobei sich die Frage stellen würde, in welcher Form diese Tags ausgewertet werden sollten. Problematisch wäre auch, dass die Spoon-Bibliothek derzeit keine Möglichkeit bietet, die Inline-Javadoc-Tags zu analysieren.
Des Weiteren könnte eine zusätzliche Konfiguration implementiert werden, die dafür sorgt, dass die von Oracle vorgeschlagene Reihenfolge der Javadoc-Tags eingehalten wird. [ORA04]
Und zu guter Letzt wäre eine Forschungsfrage interessant, die sich damit befasst, wie sich die Javadoc Code Smells zu anderen Code Smells verhalten. Ein Beispiel hierfür wäre die Prüfung, ob Gottklassen deutlich häufiger von Javadoc Code Smells betroffen sind als andere Klassen. Analog funktioniert dieses Beispiel auch für lange Methoden oder andere Code Smells. Leider hätten die zusätzlichen Analysen, die benötigt werden, um diese Forschungsfrage zu beantworten, den Zeitrahmen dieser Bachelorarbeit gesprengt.

Literaturverzeichnis

[Abt10] Abts, Dietmar: *Grundkurs JAVA: Von den Grundlagen bis zu Datenbank- und Netzanwendungen*. 6., erweiterte Auflage. Wiesbaden : Vieweg+Teubner Verlag / Springer Fachmedien Wiesbaden GmbH, 2010. – S. 13

[BH14] Burger, Stefan ; Hummel, Oliver: *Über die Auswirkungen von Refactoring auf Softwaremetriken*. In: Gesellschaft für Informatik, 2014

[Bos18] Bosch, Artur: *Priorisierung von Code Smells*. Universität Bremen, Masterarbeit, 2018

[Bra16] Brack, Fagner: *Code Comment Is A Smell: Do not use code comments to fix a badly written code*. URL: `https://medium.com/@fagnerbrack/code-comment-is-a-smell-4e8d78b0415b`, 2016. – Zuletzt abgerufen am: 09. Oktober 2019

[Die01] Dieterich, Ernst-Wolfgang: *Java 2: Von den Grundlagen bis zu Threads und Netzen*. 2., überarb. Aufl. München : Oldenbourg, 2001. – S. 5–9

[FBB+99] Fowler, Martin ; Beck, Kent ; Brant, John ; Opdyke, William ; Roberts, Don: *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999 (The Addison-Wesley object technology series). – S. 9, S. 63–72

[Fow06] Fowler, Martin: *CodeSmell*. URL: `https://martinfowler.com/bliki/CodeSmell.html`, 2006. – Zuletzt abgerufen am: 06. Oktober 2019

[GJS+15] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad ; Buckley, Alex: *The Java® Language Specification: Java SE 8 Edition*. URL: `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`, 2015. – Zuletzt abgerufen am: 24. Oktober 2019

[Gno16] Gnoyke, Harm: *ISO, weshalb warum? Ist Software-Qualität Geschmackssache?* URL: `https://www.embarc.de/software-qualitaet-iso-25010/`, 2016. – Zuletzt abgerufen am: 01. Oktober 2019

[Ham18] Hamedy, Rafiullah: *A short summary of Java coding best practices: based on coding standards by Oracle, Google, Twitter and Spring Framework*. URL: `https://medium.com/@rhamedy/a-short-summary-of-java-coding-best-practices-31283d0167d3`, 2018. – Zuletzt abgerufen am: 09. Oktober 2019

[Hem11] Hemmerich, Wanja ; MatheGuru (Hrsg.): *Signifikanz, Signifikanzniveau.* URL: `https://matheguru.com/stochastik/signifikanz-signifikanzniveau.html`, 2011. – Zuletzt abgerufen am: 27. Oktober 2019

[Kos17] Koschke, Rainer: *Vorlesung zu Software Analytics.* 2017

[LL14] Liguori, Robert ; Liguori, Patricia: *Java 8 pocket guide.* Beijing and Cambridge : O'Reilly Media, 2014. – Kap. 4

[Mar13] Martin, Robert C.: *Clean Code - Refactoring, Patterns, Testen und Techniken für: Deutsche Ausgabe.* Frechen : MITP, 2013 (mitp Professional). – Kap. 4, Kap. 17.1

[Mei09] Meiert, Jens O.: *Einfühung in Wartbarkeit.* URL: `https://meiert.com/de/publications/articles/20090907/`, 2009. – Zuletzt abgerufen am: 01. Oktober 2019

[MVL03] Mäntylä, Mika ; Vanhanen, Jari ; Lassenius, Casper: *A Taxonomy and an Initial Empirical Study of Bad Smells in Code.* In: Proceedings of the International Conference on Software Maintenance, 2003

[ORA93a] ORACLE (Hrsg.): *How and When To Deprecate APIs.* URL: `https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/deprecation.html#how`, 1993. – Zuletzt abgerufen am: 24. Oktober 2019

[ORA93b] ORACLE (Hrsg.): *javadoc - The Java API Documentation Generator.* URL: `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html`, 1993. – Zuletzt abgerufen am: 06. Oktober 2019

[ORA04] ORACLE (Hrsg.): *How to Write Doc Comments for the Javadoc Tool.* URL: `https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#orderoftags`, 2004. – Zuletzt abgerufen am: 24. Oktober 2019

[PMD19] PMD (Hrsg.): *Dokumentation.* URL: `https://pmd.github.io/pmd-6.18.0/`, 2019. – Zuletzt abgerufen am: 06. Oktober 2019

[PMP+06] Pawlak, Renaud ; Monperrus, Martin ; Petitprez, Nicolas ; Noguera, Carlos ; Seinturier, Lionel: *Spoon: Program Analysis and Transformation in Java.* In: HAL-Inria, 2006

[PMP+16] Pawlak, Renaud ; Monperrus, Martin ; Petitprez, Nicolas ; Noguera, Carlos ; Seinturier, Lionel: *SPOON : A library for implementing analyses and transformations of Java source code.* In: Software: Practice and Experience, 2016

[PP19] Petri, Britta ; Petri, Björn: *Java-Tutorial.org - Java lernen leicht gemacht: Vererbung*. URL: https://www.java-tutorial.org/vererbung.html, 2019. – Zuletzt abgerufen am: 23. Oktober 2019

[Sch10] Schneider, Stefan: *Dokumentieren von Javaprogrammen (javadoc)*. URL: http://scalingbits.com/java/javakurs1/javadoc, 2010. – Zuletzt abgerufen am: 07. Oktober 2019

[Sch19] Schulz, Dominique: *Evolution von Code Smells*. Universität Bremen, Bachelorarbeit, 2019

[Sou17] Sourour, Bill: *Putting comments in code: the good, the bad, and the ugly.* URL: https://www.freecodecamp.org/news/code-comments-the-good-the-bad-and-the-ugly-be9cc65fbf83/, 2017. – Zuletzt abgerufen am: 09. Oktober 2019

[Ste16] Stephanie ; Statistics How To (Hrsg.): *Mann Kendall Trend Test: Definition, Running the Test*. URL: https://www.statisticshowto.datasciencecentral.com/mann-kendall-trend-test/, 2016. – Zuletzt abgerufen am: 27. Oktober 2019

[Ste18] Steinbeck, Marcel: *LibVCS4j: A Java Library for Repository Mining*. URL: http://pi.informatik.uni-siegen.de/gi/stt/38_2/01_Fachgruppenberichte/WSRE_2018/21_WSRE2018_paper_12.pdf, 2018. – Zuletzt abgerufen am: 06. Oktober 2019

[Ste19] Steinbeck, Marcel: *LibVCS4j: Readme*. URL: https://github.com/uni-bremen-agst/libvcs4j, 2019. – Zuletzt abgerufen am: 06. Oktober 2019

[t2i] t2informatik (Hrsg.): *Code-Smell*. URL: https://t2informatik.de/wissen-kompakt/code-smell/, . – Zuletzt abgerufen am: 06. Oktober 2019

[Ull12] Ullenboom, Christian: *Java ist auch eine Insel: Das umfassende Handbuch*. 10., aktualisierte und überarb. Aufl., 1. Nachdr. Bonn : Galileo Press, 2012 (Galileo computing). – S. 120–122, S. 547–548, S. 616, S. 1259–1269

[Wag12] Wagner, Chris: *Java-Grundlagen: Variablen in Java – Deklaration, Sichtbarkeit und Lebensdauer*. URL: https://www.programmierenlernenhq.de/java-grundlagen-variablen-in-java-deklaration-sichtbarkeit-und-lebensdauer/, 2012. – Zuletzt abgerufen am: 24. Oktober 2019

[Zai12] Zaiontz, Charles: *Mann-Kendall Test*. URL: https://www.real-statistics.com/time-series-analysis/

`time-series-miscellaneous/mann-kendall-test/`, 2012. – Zuletzt abgerufen am: 27. Oktober 2019

A. Installationshinweise

Die folgenden Installationshinweise wurden aus der Sicht eines Windows-Betriebssystems geschrieben. Die Vorgehensweise für Linux oder MacOS sollte allerdings analog anwendbar sein. Für die Verwendung der LibVCS4j-Bibliothek wird Java 11 und Maven beziehungsweise Gradle benötigt. Des Weiteren muss darauf geachtet werden, dass das Teilprojekt *"libvcs4j-tools"* in seiner *"build.gradle"* - Datei die Spoon Version 7.5 verwendet, da diese für eine vollständige Funktionsweise des Javadoc-Detektors benötigt wird. Außerdem muss darauf geachtet werden, dass die Google Guava - Dependence für den Evaluator im "classpath" gesetzt wurde.
Des Weiteren wird empfohlen, den Ordner *"analysis"* des USB-Sticks direkt in das Root-Verzeichnis der C-Partition zu kopieren, da so keine weiteren Änderungen in der Ausführung des Evaluators vorgenommen werden müssen. Alternativ müssen die Pfade in der Mann-Kendall-Trend-Analyse sowie in der Evaluation-Klasse geändert werden. Die Evaluation-Klasse definiert dabei die Ausführung des Evaluators, diese ist im folgenden Pfad des LibVCS4j Projektes zu finden: *"/libvcs4j-tools/src/test/java/de/unibremen/informatik/st/libvcs4j/spoon/codesmell/missingJavadoc/"*.
Zu beachten ist hierbei, dass die Methoden *"evaluateQ1"* für die erste Forschungsfrage und *"evaluateQ2ToQ3"* für die zweite und dritte Forschungsfrage verwendet werden muss, um die Ergebnisse der Analyse zu reproduzieren. Des Weiteren ist es sehr wichtig, dass ein Teilbereich der zu analysierenden Projekte ausgewählt werden muss, da ansonsten der Festplattenspeicher überlasten würde, wenn alle Projekte auf einmal analysiert werden sollen. Zur Konfiguration der zu untersuchenden Projekte wurden in den beiden Methoden jeweils ein erläuternder Kommentar gegeben. Außerdem ist es hilfreich die beiden Listen *"analysis_q1.csv"* und *"analysis_q2-q3.csv"* aus dem Verzeichnis *"analysis"* für die Konfiguration zu verwenden. Diese Listen beinhalten den Pfad zu den Projekten sowie die Unterverzeichnisse, die untersucht werden sollen, um, so wie oben beschrieben, nur den wichtigen Quellcode des jeweiligen Projektes zu analysieren. Die Ergebnisse der Analysen werden ebenfalls in dem Verzeichnis *"analysis"* in dem Unterverzeichnis *"csv"* gespeichert. Sollte das Verzeichnis nicht in dem Root-Verzeichnis der C-Partition liegen, müssen in der Klasse *"JavadocEvaluator"*, welche unter dem Pfad *"libvcs4j-tools/src/main/java/de/unibremen/informatik/st/libvcs4j/spoon/codesmell/missingJavadoc"* des libvcs4j-Verzeichnisses zu finden ist, ebenfalls Änderungen an dem Pfad, in dem die Ergebnisse gespeichert werden sollen, vorgenommen werden. Dies wird mittels der Konstanten *"PATH_TO_SAVEDIR"* konfiguriert. Zu beachten ist allerdings, dass die Evaluation aller Projekte, besonders die der ersten Forschungsfrage, sehr lange dauert.

Die Mann-Kendall-Trend-Analyse kann mittels der Datei *"mann_kendall.py"* reproduziert werden. Zu finden ist diese Datei im *"analysis"* Verzeichnis. Die Ergebnisse der Mann-Kendall-Trend-Analyse werden in der *"results.csv"* - Datei abgespeichert. Diese Datei liegt ebenfalls in dem Verzeichnis.
Wenn das *"analysis"*-Verzeichnis nicht in das oben genannte Verzeichnis kopiert wurde, müssen zusätzlich in der *"mann_kendall.py"*-Datei die Pfade in der 5. und 18. Zeile angepasst werden.

B. Liste der analysierten Projekte

In der folgenden Tabelle sind alle analysierten Projekte sowie deren untersuchten Verzeichnisse aufgeführt. Dabei werden die Projekte, die ausschließlich für die erste Forschungsfrage analysiert werden konnten blau und die Projekte, die ausschließlich für die zweite und dritte Forschungsfrage analysiert werden konnten gelb gekennzeichnet. Die ursprüngliche Liste stammt aus der Bachelorarbeit von Dominique Schulz. [Sch19]

Projekte	Unterverzeichnis
adt4j	adt4j/src
Aeron	aeron-client
Agrona	agrona/src/main
airline	src/main
almanac-converter	src/main
arangodb-java-driver	src/main
ArchUnit	archunit/src/main
async-http-client	client/src/main
aurora	src/main/java
auto	common/src/main
automon	automon/src/main
avian	classpath/avian
awaitility	awaitility/src/main
bigqueue	src/main
blade	src/main
burst	burst/src/main
caffeine	caffeine/src/main
cglib	cglib/src/main
checkstyle	src/main
Chronicle-Map	src/main
CodenameOne	CodenameOne/src
cogcomp-nlp	ner/src/main
commons-csv	src/main
concurrentunit	src/main
config	config/src/main
consul-api	src/main
cqengine	code/src/main

crawler4j	crawler4j/src/main
cucumber-jvm	core/src/main
cukes-rest	cukes-core/src/main
cyclops-react	cyclops/src/main
derive4j	processor/src/main
Dex	src/com
dozer	core/src/main
drjava	drjava/src
dropwizard-circuitbreaker	src/main
eclipse-collections	eclipse-collections/src/main
Erdos	src/main
error-prone	core/src/main
eureka	eureka-core/src/main
failsafe	src/main
fast-serialization	src/main
fastjson	src/main
faux-pas	src/main
feather	feather/src/main
feign	core/src/main
findbugs	findbugs/src/java
fixture-factory	src/main
flatbuffers	java
flexy-pool	flexy-pool-core/src/main
flyingsaucer	flying-saucer-core/src/main
FreeBuilder	src/main
freecol	src
governator	governator-core/src/main
graphhopper	core/src/main
grpc-java	core/src/main
gson	gson/src/main
guava	guava/src
guice	core/src
hdiv	hdiv-core/src/main
HikariJSON	src/main
honest-profiler	src/main/java
HotswapAgent	hotswap-agent-core/src/main
Hystrix	hystrix-core/src/main
imgscalr	src/main
j2objc	translator/src/main

j8spec	src/main
jackson-dataformat-csv	src/main
jackson-datatype-money	src/main
jacoco	org.jacoco.core/src
janala2	src/main
java8-tutorial	src
javacpp	src/main
javamelody	javamelody-core/src/main
javaparser	javaparser-core/src
javapoet	src/main
javassist	src/main
javasymbolsolver	java-symbol-solver-core/src
JavaVerbalExpressions	src/main
jbot	jbot/src/main
jcalendar	JCalendar/src
JCTools	jctools-core/src/main
jedis	src/main
jeromq	src/main
Jest	jest-common/src/main
jetcd	src/main
jfairy	src/main
jgit	org.eclipse.jgit/src
jgrapht	jgrapht-core/src/main
jgraphx	src
jHiccup	src/main
jimfs	jimfs/src/main
jitwatch	core/src/main
jjwt	impl/src/main
jna	src
jnr-ffi	src/main
jolt	jolt-core/src/main
JSAT	JSAT/src
json-io	src/main
jsonld-java	core/src/main
JsonPath	json-path/src/main
JsonSurfer	jsurfer-core
jtk	core/src/main
junit-dataprovider	core/src/main
jython	src

kaconf	src/main
keyczar	java/code/src
Koloboke	jpsg/core/src/main
kryo	src
lambda-behave	lambda-behave/src/main
languagetool	languagetool-core/src/main
lanterna	src/main
LatencyUtils	src/main
libgdx	gdx/src
logbook	logbook-core/src/main
lombok	src/core
mapsforge	mapsforge-core/src/main
mapstruct	core/src/main
MariaDB4j	mariaDB4j-core/src/main
maven-wrapper	src/main
micro-server	micro-core/src/main
minio-java	api/src/main
Mixin	src/main
moco	moco-core/src/main
modelmapper	core/src/main
modernizer-maven-plugin	modernizer-maven-plugin/src/main
moshi	moshi/src/main
MutabilityDetector	src/main
nakadi	src/main
nanohttpd	core/src/main
nbvcxz	src/main
nifty	nifty-core/src/main
Orienteer	orienteer-core/src/main
orika	core/src
oryx	app/oryx-app-common/src
owner	owner/src/main
pac4j	pac4j-core/src/main
pinpoint	collector/src/main
pmd	pmd-core/src/main
polyglot-maven	polyglot-java/src
presto	presto-main/src/main
protobuf	java/core/src/main
protonpack	src/main
quartz	quartz-core/src/main

raml-tester	src/main
randomizedtesting	randomized-runner/src/main
reactive-streams-jvm	tck/src/main
realm-java	realm/realm-library/src/main
redisson	redisson/src/main
requery	requery/src
resilience4j	resilience4j-core/src/main
rest-assured	rest-assured/src/main
rest.li	restli-common/src/main
RestExpress	core/src/main
ribbon	ribbon-core/src/main
riptide	riptide-core/src/main
rocketmq	common/src/main
selma	selma/src
service-proxy	core/src/main
simple-binary-encoding	sbe-tool/src/main/java
siren4j	src/main
Smack	smack-core/src/main
soot	src/main/java
spatial4j	src/main
speedment	generator-parent/generator-translator/src/main
spoon	src/main
spring-boot	spring-boot-project/spring-boot/src/main
stagemonitor	stagemonitor-core/src/main
streamex	src/main
Sysmon	src
tablesaw	core/src/main
tape	tape/src/main
tess4j	src/main
testcontainers-java	core/src/main
thumbnailator	src/main
Time4J	base/src/main
tracer	tracer-core/src/main
truth	core/src/main
TwelveMonkeys	common/common-io/src/main
typetools	src/main
underscore-java	src/main
univocity-parsers	src/main
urnlib	src/main

wire	wire-schema/src/main
zxing	core/src/main

Tabelle B.1.: Liste der analysierten Projekte