Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Stile von API-Dokumentationen anhand von Java und Python

Ahmet-Serdar Karakaya
Matrikelnummer: 4552318
ahmetserdar@hotmail.de

Betreuer und Gutachter: Prof Dr. Lutz Prechelt
Zweitgutachterin: Prof. Dr. Claudia Müller-Birn

Berlin, 11.05.2017

Zusammenfassung

APIs bieten komplexe Funktionalitäten, die von Softwareentwicklern in die Software, an der sie gerade arbeiten, eingebaut werden können. Das macht APIs für die Softwareentwicklung sehr wichtig. APIs sind aber auch komplex und deshalb sind API-Dokumentationen sehr wichtig. Laut Studien ist der allgemeine Zustand von API-Dokumentationen schlecht, was oft zu Schwierigkeiten beim Erlernen und Nutzen von APIs führt. Mit einer Charakterisierung von Stilen von API-Dokumentationen und der Herausarbeitung von Vor- und Nachteilen einzelner Aspekte dieser Stile, können bessere Anleitungen für gute API-Dokumentationen erstellt werden. Im Rahmen dieser Masterarbeit werden die Dokumentationen der Java- und der Python-API untersucht und Stileigenschaften bzw. Dokumentationsstile vorgestellt und somit ein wichtiger Beitrag dazu geleistet, Anleitungen für gute API-Dokumentationen zu schreiben.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder Ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

11.05.2017

Ahmet-Serdar Karakaya

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Aufbau der Arbeit . . . 2
- 1.2 Sprachgebrauch . . . 3

2 API und Dokumentation **4**
- 2.1 Wiederverwendbarkeit . . . 4
- 2.2 API . . . 5
- 2.3 API-Dokumentation . . . 8

3 Aktueller Stand der Forschung **10**
- 3.1 API-Usability . . . 10
- 3.2 API-Dokumentation . . . 12
- 3.3 Wissenstypen in API-Dokumentationen . . . 22

4 Forschungsmethoden **29**
- 4.1 Typisierung der Dokumentationen . . . 29
- 4.2 Grounded Theory Methodology . . . 34
- 4.3 Gruppendiskussion mit Python-Entwicklern . . . 36
- 4.4 E-Mailverkehr der Autoren . . . 37

5 Dokumentationsstile und Stileigenschaften **39**
- 5.1 Aufbau und Form der Dokumentation . . . 40
- 5.2 Beispielnutzung . . . 45
- 5.3 Lokalität . . . 49
- 5.4 Beziehungssichtbarkeit . . . 55
- 5.5 Wissenstypenordnung . . . 58
- 5.6 Zusammenfassung . . . 62
  - 5.6.1 Referenzstil . . . 62
  - 5.6.2 Buchstil . . . 62
  - 5.6.3 Referenzstil vs. Buchstil . . . 67

6 Fazit **68**
- 6.1 Zusammenfassung . . . 68
- 6.2 Ausblick . . . 69

A Anhang **i**
- A.1 Ergebnisse der Studie von Zibran et al. (2011) . . . i
- A.2 Ergebnisse der Studie von Grill et al. (2012) . . . ii

1 Einleitung

„Das Rad neu erfinden“ - Diese im Volksmund weit verbreitete Redewendung drückt aus, etwas Bekanntes neu zu entwickeln oder sich an längst gelösten Problemen abzuarbeiten. Auch in der Softwareentwicklung bietet es sich oft an etwas altes, bereits Bestehendes neu zu entwickeln. Es kann manchmal sinnvoll sein, ein Problem mit einer bereits existierenden Lösung anzugehen und es komplett zu lösen. Allerdings ist es nicht garantiert, dass sich der Aufwand dafür lohnt, weil die neue Lösung nicht besser ist und der Mehrwert der neuen Lösung den Aufwand nicht entschädigt.
Eine Ingenieurtätigkeit zeichnet sich dadurch aus, bereits bestehende wissenschaftliche Lösungen für aktuelle praktische Probleme anzuwenden. Das ist ein bewährter Ansatz, um Probleme zu lösen, und ist auch ein wichtiger Bestandteil der Softwaretechnik.
Die immer höheren Anforderungen der Gesellschaft und der Industrie erfordern immer komplexere Software und somit wird auch der Entwurf bzw. die Architektur moderner Software immer komplexer.
Das macht eine Sammlung von Lösungsschablonen für immer wiederkehrende Entwurfsprobleme sehr wichtig. Dafür gibt es in der Softwaretechnik die Entwurfsmuster. Sie beschreiben Entwurfsprobleme und liefern Ideen dafür, wie sie gelöst werden können.
Dadurch können Softwareentwickler bei Entwurfsproblemen auf die Lösungsschablonen der Entwurfsmuster zurückgreifen, diese für ihr Problem anpassen und anwenden. Das ermöglicht und vereinfacht die Realisierung von komplexen Softwareprodukten und sorgt für eine höhere Qualität.
Eine weitere Möglichkeit vorhandene Lösungen wiederzuverwenden werden von sogenannten APIs bereitgestellt. API steht für Application Programming Interface und ist eine Programmierschnittstelle zu Softwarebibliotheken, die von Softwareentwicklern genutzt werden kann, um ihre eigene Software zu entwickeln. Dazu werden Funktionalitäten, die die API bietet, in die eigene Software eingebaut, wodurch sie nicht selber entwickelt werden müssen.
Neben Softwareprodukten werden auch APIs immer komplexer. Das macht eine Dokumentation der APIs immer wichtiger, um eine korrekte Verwendung durch den Softwareentwickler, der die API nutzt, zu gewährleisten.
Eine gute API-Dokumentation zu schreiben ist aber nicht leicht, weil die APIs komplex sind und weil es so etwas wie Entwurfsmuster für API-Dokumentationen bisher nicht gibt.
Damit könnten bessere API-Dokumentationen erstellt werden, was dazu führen würde, dass Softwareentwickler die genutzten APIs besser verstehen. Folglich würde das die Softwareentwicklung begünstigen und für bessere Software bzw. kürzere Entwicklungszeiten sorgen.
Um solche Entwurfsmuster für API-Dokumentationen entwickeln zu können, müssen aber viele Fragen geklärt werden. Zum einen ist nicht ganz klar, was

unter einer „guten API-Dokumentation “ genau zu verstehen ist, wie sie aussieht und wie sie erstellt werden kann.
Obwohl APIs schon lange zum Alltag von Softwareentwicklern gehören, sind diese Fragen noch offen. Denn dafür, dass es sich schon um ein älteres Thema handelt, sind API-Dokumentationen wenig erforscht worden.
Ein möglicher Grund könnte sein, dass API-Dokumentationen von Menschen in natürlicher Sprache geschrieben werden und deswegen eine Analyse schwer ist. Außerdem ist es schwer gut verallgemeinerbare Experimente durchzuführen, die API-Dokumentationen untersuchen sollen.
Um die Forschung in diesem Gebiet voranzutreiben und Entwurfsmuster von API-Dokumentationen erstellen zu können, sind zum einen neue Fachbegriffe notwendig, um sich besser über die Inhalte von API-Dokumentation verständigen zu können, und zum anderen muss eine Charakterisierung von Stilen von API-Dokumentationen vorgenommen werden.
Maalej und Robillard haben eine Taxonomie von Wissenstypen, die in Dokumentationen vorkommend, vorgestellt [1]. Damit haben sie Fachbegriffe eingeführt, die verwendet werden können, um sich besser über die Inhalte von API-Dokumentationen verständigen zu können.
Was aber noch ansteht ist eine Charakterisierung von Stilen. Ziel dieser Masterarbeit ist es ...

1. ... zu klären, ob es so etwas wie unterschiedliche Stile von API-Dokumentationen gibt.
2. ... einige Eigenschaften, die einen Stil ausmachen, zu identifizieren.
3. ... anhand dieser Eigenschaften zwei unterschiedliche Stile zu beschreiben.
4. ... Methoden zu entwickeln, die das Erreichen dieser Ziele vereinfachen bzw. ermöglichen.

1.1 Aufbau der Arbeit

Die vorliegende Masterarbeit ist wie folgt gegliedert:
In Kapitel 2 werden die Begriffe *Wiederverwendbarkeit*, *API* und *API-Dokumentation* erklärt und näher erläutert. Es wird darauf eingegangen was Wiederverwendbarkeit ist, wie es mit APIs erreicht werden kann und welche Rolle API-Dokumentationen dabei spielen.
In Kapitel 3 wird der aktuelle Stand der Forschung dargelegt. Es werden Studien vorgestellt, die die Relevanz der API-Usability untersuchen, welche Einflussfaktoren die API-Usability hat und welche Rolle die API-Dokumentation für die API-Usability spielt. Außerdem wird die bereits angesprochene Studie von Maalej und Robillard vorgestellt, sowie eine Folgestudie, die von Wilderman durchgeführt wurde und für diese Masterarbeit ebenfalls wichtig ist.

In Kapitel 4 werden die Forschungsmethoden vorgestellt, die für das Erreichen der Ziele dieser Masterarbeit entwickelt bzw. verwendet wurden. Dazu gehört das Typisieren von Teilen der Java- und Python-API-Dokumentation mit den Wissenstypen von Maalej und Robillard, um inhaltliche Unterschiede besser zu erkennen und auf Stileigenschaften zu kommen. Außerdem wird ein Versuch beschrieben die Grounded Theory Methodologie zu verwenden, die eine in den Sozialwissenschaften bewährte Forschungsmethode zum Generieren von Theorien ist. Zusätzlich werden Erfahrungen mit E-Mail-Verteilern und Gruppendiskussionen beschrieben, die Einsichten über Denkweisen von Autoren und Nutzern von API-Dokumentationen gaben.
In Kapitel 5 werden die Ergebnisse der Forschung, die im Rahmen dieser Masterarbeit durchgeführt wurde, vorgestellt. Es werden fünf Stileigenschaften benannt sowie die Dokumentationsstile *Referenzstil* und *Buchstil*, deren Stileigenschaften unterschiedlich ausgeprägt sind. Die Stileigenschaften umfassen *Aufbau und Form*, *Beispielnutzung*, *Lokalität*, *Beziehungssichtbarkeit* und *Wissenstypordnung*.
In Kapitel 6, dem letzten Kapitel, wird die Masterarbeit zusammengefasst und ein Ausblick über weitere Schritte zur Charakterisierung von Dokumentationsstilen und weiterer Stileigenschaften gegeben.

1.2 Sprachgebrauch

Auf geschlechtsneutrale Formulierungen wurde aus Gründen der Lesbarkeit verzichtet. Im Text sind immer beiderlei Geschlechter gemeint.
Ferner werden an manchen Stellen Anglizismen und englische Zitate verwendet, wenn sonst die Bedeutung verfälscht werden würde.

2 API und Dokumentation

In dieser Arbeit spielen APIs und API-Dokumentationen eine große Rolle. Deshalb wird in diesem Kapitel erklärt, was unter diesen Begriffen zu verstehen ist. Zunächst geht es um ein wichtiges Thema der Softwaretechnik: Die Wiederverwendbarkeit. Unter diesem Konzept ist das Benutzen von bereits existierenden Softwareartefakten im aktuellen Softwareprojekt zu verstehen. Anschließend wird auf APIs eingegangen, die ein sehr verbreitetes Mittel sind, um Wiederverwendbarkeit zu erreichen. Denn es handelt sich dabei um Schnittstellen von Softwarebibliotheken, über die die Funktionalitäten der Softwarebibliothek genutzt werden können.
Da APIs sehr groß und komplex sein können, ist es sehr wichtig, dass es Dokumentationen gibt. In einer Dokumentation wird idealerweise erklärt, was die API im Allgemeinen macht, aus welchen Elementen sie besteht, welchen Zweck die jeweiligen Elemente erfüllen und welche Beziehung sie zueinander haben.

2.1 Wiederverwendbarkeit

Der im Softwaremarkt vorherrschende Konkurrenzdruck zwingt Softwareunternehmen zu immer kürzeren Time-To-Markets ihrer Produkte. Mit einer früheren Markteinführung kann der Verkauf früher gestartet und der Konkurrenz zuvorgekommen werden. Davon werden mehr Verkäufe und somit auch höhere Gewinne erwartet. Damit diese aber nicht ausbleiben, muss sichergestellt werden, dass die Qualität unter der verkürzten Produktionszeit nicht allzu sehr leidet. Da ist es wichtig darauf zu achten, wie die kürzere Produktionszeit erreicht wird. Wird etwa die Betaphase, in der dem Produkt der Feinschliff verpasst wird und Fehler korrigiert werden, gestrichen, kann das die Qualität der Software erheblich verringern.
Das Prinzip der Wiederverwendbarkeit kann hier Abhilfe schaffen. Die Wiederverwendbarkeit bzw. die Wiederverwendung von Software beschreibt einen Prozess in der Softwareentwicklung, bei dem bereits existierende Softwareartefakte verwendet werden, anstatt sie neu zu erstellen [2] . Hierfür bietet es sich an, beim Programmieren auf bereits bestehende Lösungen zurückzugreifen, anstatt immer wieder das sprichwörtliche Rad neu zu erfinden.
Allerdings ist der anfängliche Aufwand für die Erstellung von Softwareartefakten, die wiederverwendbar sein sollen, höher. Der Grund ist, dass der Code abstrakter sein muss und somit schwerer zu schreiben ist. Werden diese Teile der Software in der Zukunft aber öfter verwendet, spart der Programmierer bei jeder erneuten Verwendung Zeit und dadurch amortisiert sich der Aufwand, da Wiederverwendung zu einer Kostensenkung [3] bei gleichzeitiger Verringerung der Defektdichte [4] führt.
Der abstraktere Code erlaubt eine stärkere Anpassung an die jeweils aktuellen Anforderungen und ermöglicht so einen breiteren Einsatz. Das macht

es auch für andere Softwareentwickler attraktiv ihn zu benutzen und es entstehen Sammlungen von Softwareartefakten bzw. Klassen und Funktionen. Diese Sammlungen enthalten meistens thematisch zusammenhängende Elemente und werden Programmbibliothek (oder kurz: Bibliothek, englisch: library) genannt. Solche Bibliotheken können dann von anderen Softwareentwicklern in ihre Software eingebunden und genutzt werden, sofern sie Zugang dazu haben. Um die Nutzung dieser Bibliotheken zu vereinfachen, gibt es sogenannte Programmierschnittstellen (englisch: application programming interface, kurz: **API**).

2.2 API

Die „Software Engineering Institute “[1] hat API wie folgt definiert:

> *Application Programming Interface (API) is an older technology that facilitates exchanging messages or data between two or more different software applications. API is the virtual interface between two interworking software functions, such as a word processor and a spreadsheet. This technology has been expanded from simple subroutine calls to include features that provide for interoperability and system modifiability in support of the requirement for data sharing between multiple applications. An API is the software that is used to support system-level integration of multiple commercial-off-the-shelf (COTS) software products or newly-developed software into existing or new applications. APIs are also a type of Middleware that provide for data sharing across different platforms;*[...]

Das ist eine sehr eingrenzende Definition. Sie stellt die API als ein Mittel dar, um Daten zwischen Software auszutauschen. Das mag für manche Forschungsbereiche und Taxonomien korrekt und ausreichend sein, spiegelt aber nicht den gängigen Gebrauch wider [5]. Viel treffender finde ich die Definition von de Souza et al.:
„*[...] any well-defined interface that defines the service that one component, module, or application provides to other software elements*“ [6]
Diese Definition ist viel breiter und erlaubt es ein Interface für ein Objektmodell (z.B. in Java), ein Framework, eine Softwarebibliothek, einen Toolkit oder einen Software Development Kit (SDK) als API zu bezeichnen. Auch in dieser Arbeit werden diese Dinge unter dem Begriff API zusammengefasst.
APIs werden immer häufiger verwendet. Heutzutage gibt es kaum noch Software, bei deren Produktion komplett auf APIs verzichtet wird, was an vielen Vorteilen, die die Nutzung einer API mit sich bringt, liegt [7].

[1] http://www.sei.cmu.edu/(letzter Zugriff: 10.05.2017)

Neben der Wiederverwendbarkeit gibt es ein weiteres sehr wichtiges Prinzip in der Softwaretechnik und insbesondere des objektorientierten Paradigmas, das durch den Einsatz APIs umgesetzt werden kann: Das Geheimnisprinzip (englisch: information hiding).
Es besagt, dass Softwaremodule voneinander so wenig wie möglich, so viel wie nötig wissen sollen [8]. Durch das Bereitstellen von Schnittstellen werden Abhängigkeiten zwischen der Anwendung der API und der Implementierung der API verringert. Dadurch kann die Implementierung der API verändert werden, ohne dass der Code, der auf die API zugreift, geändert werden muss.
APIs werden heutzutage in fast allen Softwareprojekten verwendet. Viele Programmierer schätzen APIs, weil diese komplexe Funktionalitäten anbieten, die sonst der Programmierer selbst hätte programmieren müssen [9]. Hinzu kommt, dass die Anforderungen der Nutzer immer steigen und Software immer komplexer wird. Diese beiden Faktoren führen dazu, dass Software zu großen Teilen aus zusammengesteckten Elementen verschiedener APIs besteht [10].
Die Nutzung von APIs kann neben den Vorteilen auch einige Nachteile mit sich bringen. Wenn eine Software auf eine API aufbaut und diese API aber nicht mehr gepflegt bzw. aktualisiert wird, stellt das ein Problem für die Wartung der Software dar.
Wird z.B. eine Sicherheitslücke in einer API entdeckt, deren Support eingestellt ist, muss die Software, die diese API benutzt, ebenfalls geändert werden. Es kann sein, dass die API open source ist und deswegen von den Entwicklern der Software aktualisiert werden kann. In beiden Fällen kann der Aufwand für die Nutzer einer API groß sein. Eine weitere Variante dieses Problems kann entstehen, wenn die API noch gepflegt und aktualisiert wird, sich aber in der neuen Version die Nutzungsbedingungen ändern. Das bedeutet, dass der Nutzer der API sich entweder an die neuen Nutzungsbedingungen anpassen muss, was nicht immer möglich ist, oder die API wechseln muss.
Wie bereits erwähnt kann das Einbinden von APIs die Testphase in der Softwareentwicklung verkürzen. Das liegt daran, dass gerade populäre APIs ausgiebig getestet wurden und die Fehler korrigiert sind. Es kann aber immer noch Bugs geben, die noch nicht entdeckt wurden. In so einem Fall ist es unter Umständen sehr schwer, den Fehler auf die API einzugrenzen und ihn anschließend zu beheben. Auch hier kann wieder eine andere API verwendet werden aber das wäre sehr aufwändig.
Abbildung 1 zeigt, wie sich die Anzahl der Web-APIs[2] von 2005 bis 2012 vervielfacht hat. Dieser Trend ging bzw. geht weiter, denn die Anzahl der Web-APIs, die auf www.programmableweb.com registriert sind, betrug im Mai 2017 ca. 17.000. Die große und immer weiter steigende Anzahl an APIs

[2]Web-APIs sind APIs deren Implementierung auf Servern sind und die per HTTP-Anfragen angesprochen werden können

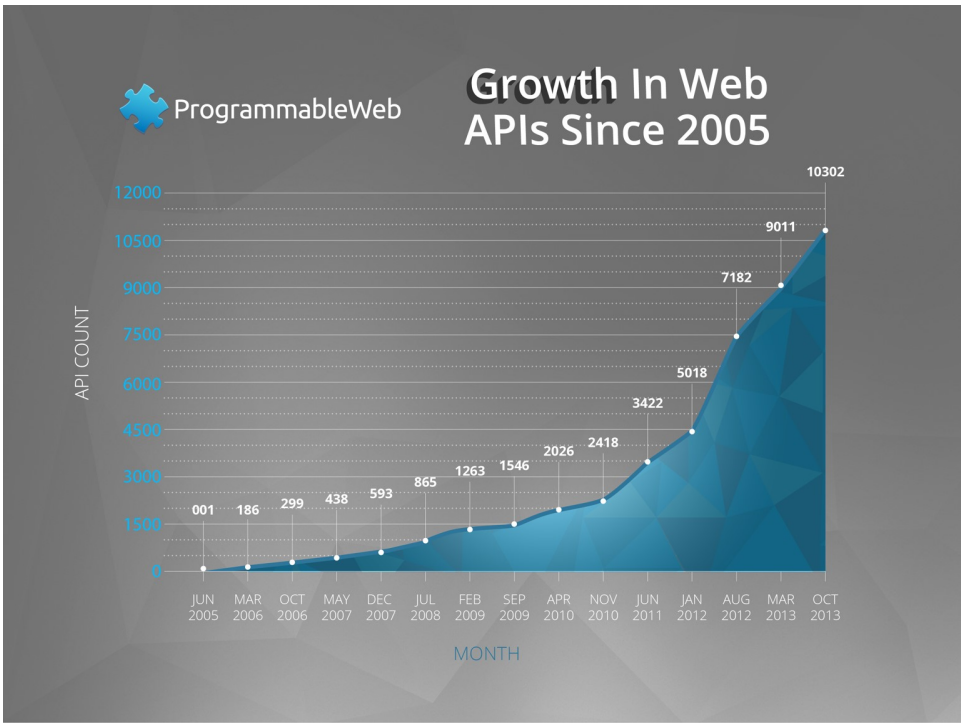


Abbildung 1: Entwicklung der Anzahl der Web-APIs.
[Quelle: https://www.programmableweb.com/api-research
(letzter Zugriff: 10.05.2017)]

kann neben der Abhängigkeit von einer API ein weiterer Nachteil werden. Es ist nämlich nicht leicht zu beurteilen, ob eine API für einen bestimmten Zweck geeignet ist oder nicht. Oft kommen für eine Anwendung mehrere APIs in Frage. Da muss der Nutzer diese APIs erst einmal finden und dann entscheiden, welcher dieser APIs für ihn in Frage kommt. Dieser Prozess kann aufwändig sein und viel Zeit in Anspruch nehmen.
Es gibt also viele APIs, von denen sich ein Softwareentwickler eine oder mehrere aussuchen kann. Das bringt die APIs in eine Konkurrenzsituation. Unabhängig davon, ob eine API von einer Firma betrieben wird, um Profit zu erzielen, oder ob sie von einer Open-Source-Gemeinde betrieben wird, muss es genug Nutzer haben, um fortbestehen zu können. Aus diesem Grund müssen APIs die Nutzer möglichst gut ansprechen. Wie kann das gelingen? „Normale“ Software, also Software, die für jedermann gedacht ist, wie z.B. Office-Programme, haben ja ein ähnliches Problem. Da hat sich die Benutzerfreundlichkeit (englisch: Usability) als ein wichtiges Merkmal herausgestellt. Zusammengefasst ist unter der Benutzerfreundlichkeit einer Software bzw. API zu verstehen, wie leicht die Benutzung erlernt werden kann und wie intuitiv die Bedienung der Software ist. Es kommen weitere

Faktoren hinzu, wie z.B. die Stabilität der Software bzw. API, die visuelle Präsentation und viele mehr.
Eine unzureichene Benutzerfreundlichkeit der API kann dazu führen, dass die API unbeliebt wird [10] und der API-Nutzer zu einer anderen API wechselt [11].

2.3 API-Dokumentation

Mehrere Studien deuten darauf hin, dass die API-Dokumentation einer der Faktoren ist, die der Benutzerfreundlichkeit von APIs am stärksten schaden [12–15].
API-Dokumentationen sind Dokumente, die die Nutzung der API erklären. Dabei ist es wichtig, dass auf alle Elemente der API eingegangen wird und dass deren Zweck, Funktions- und Nutzungsweise und Beziehungen zueinander erklärt wird.
Da es sich bei APIs um komplexe Konstrukte handelt, sind Dokumentationen wichtig, um den Durchblick zu wahren. API-Dokumentationen erfüllen idealerweise mehrere Zwecke.
Sie geben einen Überblick über die Funktionalitäten einer API. Das ist für jemanden wichtig, der für sein Projekt auf der Suche nach einer API ist und überprüfen möchte, ob eine gegebene API für seine Zwecke sinnvoll ist.
Außerdem beschreiben API-Dokumentationen die Bestandteile der API (Klassen, Funktionen, ...). Das ist vermutlich die wichtigste Funktion einer API-Dokumentation. Wenn ein API-Nutzer die Funktionalität eines Elements der API wissen möchte, kann er in den Quellcode schauen, im Internet recherchieren bzw. nachfragen, oder in der Dokumentation nachschauen. Im Quellcode nachschauen kann aufwändig und schwer sein. Und, wenn es sich um einen komplizierten Algorithmus handelt, ist es schwer die Funktionalität des Elements zu verstehen. Im Internet zu recherchieren bzw. nachzufragen kann ebenfalls problematisch sein, wenn es um eine sehr spezielle API geht, die wenige Nutzer hat, oder, wenn die gelieferten Lösungsvorschläge schlecht sind und andere Probleme verursachen. In diesem Fall bleibt die API-Dokumentation die beste Möglichkeit, um über die Funktionalität eines API-Elements zu erfahren.
Ein weiterer wichtiger Zweck, den APIs erfüllen, ist es, dass sie idealerweise den API-Nutzern erklären, wie sie bestimmte Aufgaben mit der API lösen. Dazu sind Code-Beispiele sinnvoll, die ein bestimmtes Szenario umsetzen.
Früher waren API-Dokumentationen meistens auf Papier gedruckt, während sie heutzutage fast nur in digitaler Form vorhanden sind [16]. Die Vorteile digitaler Dokumentationen liegen auf der Hand: digitale API-Dokumentation können mit wenig Aufwand aktualisiert oder korrigiert werden, während API-Dokumentationen auf gedrucktem Papier veralten und neu gedruckt werden müssen.

Darüber hinaus bieten die meisten Programme, mit denen eine API-Dokumentation betrachtet werden kann, Suchfunktionen. Diese können gerade bei großen API-Dokumentationen, deren Elemente nicht alphabetisch sortiert sind, sehr hilfreich sein.
Eine weitere Hilfe sind Hyperlinks, die beim Anklicken auf eine bestimmte Stelle der API-Dokumentation oder einer externen Seite führen.
Trotz dieser Hilfen werden API-Dokumentation von vielen Softwareentwicklern als die größte Hürde beim Lernen einer API betrachtet [14].
Auch wenn einige Ursachen dieses Problems (siehe Kapitel 3) aufgedeckt wurden, so ist es immer noch nicht klar, wie eine gute API-Dokumentation aussieht und wie sie am besten zu erstellen ist. Aus diesem Grund ist weitere Forschung auf dem Gebiet notwendig.
Das sind wichtige Aufgaben, die API-Dokumentationen haben. Darum ist die Forschung in diesem Bereich wichtig, damit herausgefunden werden kann, wie API-Dokumentationen diese Aufgaben möglichst gut erfüllen können.

3 Aktueller Stand der Forschung

Eine sehr ausführliche Literaturstudie zu API-Usability ist in der Doktorarbeit von Björn Kahlert [17] zu finden. In dieser Literaturstudie werden viele Studien über APIs, API-Dokumentationen und API-Usability vorgestellt und dabei auf die Studien stärker eingegangen, die für Doktorarbeit von hoher Relevanz sind.
In diesem Kapitel werden manche Studien davon aus einer anderen Perspektive betrachtet und auf die Studien eingegangen, die für diese Arbeit von hoher Bedeutung sind. Außerdem werden weitere Studien vorgestellt, von denen manche nach der Doktorarbeit von Kahlert entstanden sind, um einen aktuelleren Forschungsstand darzulegen.
Es wird zunächst auf den Forschungsstand der API-Usability eingegangen. Danach werden einige Studien zu API-Dokumentationen vorgestellt, die die Relevanz von API-Dokumentationen für Softwareentwickler klären. Es folgen Studien, die Probleme von API-Dokumentationen identifizieren und anschließend Studien, die diese Probleme untersuchen und Lösungsvorschläge vorstellen. Zum Schluss wird im Besonderen auf die für diese Arbeit sehr relevanten Studien eingegangen.

3.1 API-Usability

Bis vor einigen Jahren gab es sehr wenige Studien über API-Usability [18]. Seit 2010 wird in diesem Bereich wieder mehr Forschung betrieben. Laut der ACM Digital Library[3] gab es bis 2010 sieben Studien, die „API usability“ erwähnt haben und seitdem sind 13 weitere hinzugekommen. Zuvor hat sich die API-Forschung eher darauf konzentriert die Mächtigkeit [19] und Modularität [20] von APIs zu steigern.
Der Grund für einen Anstieg der Forschungsbemühungen bezüglich API-Usability liegt vermutlich daran, dass APIs immer zahlreicher und immer größer werden [5]. Dennoch gibt es dafür, dass API-Usability so wichtig ist, sehr wenige Studien darüber [7].
Es ist schwierig die Anfänge der API-Usability-Forschung zu finden, da zuvor andere Begriffe verwendet wurden. Hinzu kommt, dass in manchen Fällen die API-Usability nur Teil des Forschungsgegenstands waren.

API-Usability als Problem

Als **Rosson und Carroll** 1996 untersuchten welche Strategien Smalltalk-Programmierer benutzen, um Code wiederzuverwenden, sind sie auf eine Art API-Usabilityproblem gestoßen [21]. Sie haben eine Studie mit vier Smalltalk-Experten durchgeführt, in der sie eine Reihe von Aufgaben mit dem Einsatz von Software-Bibliotheken lösen sollten.

[3] http://www.dl.acm.org/(letzter Zugriff: 10.05.2017)

Dabei stellten sie unter anderem fest, dass Programmierer gerne Gebrauch von ausführlichen Code-Beispielen machen, indem sie diese in ihren Code kopieren und ihren Anforderungen anpassen. Diesem Vorgang gaben sie die Bezeichnung „debugging into existence“, der auch in darauffolgenden Studien übernommen wird [22]. Allerdings gab es Schwierigkeiten die relevanten Code-Beispiele als solche zu erkennen. Die Autoren der Studie haben daraus den Schluss gezogen, dass API-Entwickler beim Entwurf der API und der API-Dokumentation die Wiederverwendbarkeit aus Sicht des API-Nutzers beachten sollten, um die APIs benutzerfreundlicher zu machen.

API-Usability - Einflussfaktoren und Probleme

McLellan et al. haben die Rolle von Beispielen beim Entwerfen einer API untersucht [23]. Die Idee dabei ist, dass ein Produkt besser entworfen wird, wenn beim Entwurf die Nutzung durch den Kunden im Mittelpunkt steht. Dabei werden eine oder mehrere Szenarien genommen und die API mit dem Ziel entworfen, für dieses Szenario möglichst hilfreich zu sein.
Sie konnten beobachten, dass so eine entworfene API besser benutzt werden kann, wenn sie so genutzt wird, dass es den Beispielszenarien ähnelt. Wenn die API aber in ganz anderen Szenarien genutzt wird, tauchen Probleme auf und die Nutzer der API haben Schwierigkeiten sie ihren Wünschen entsprechend zu nutzen.
Duala-Ekoko und Robillard haben eine Studie mit der Fragestellung, welche Fragen API-Nutzer beantworten müssen, die eine API zum ersten Mal benutzen, durchgeführt [24]. Dazu haben sie einen Kontrollversuch (Controlled Experiment) mit 20 Teilnehmern durchgeführt. Jeder Teilnehmer musste alleine in einer Programmiersitzung zwei Aufgaben lösen, wobei er für jede Aufgabe eine andere API nutzen sollte. Die Sitzung wurde mit einer Kamera und einem Mikrofon aufgezeichnet und der Programmierer wurde angewiesen die Think-Aloud-Methode zu verwenden, bei dem er seine Gedanken in Worte fassen muss. Außerdem wurden Interviews mit den Teilnehmern geführt. Sie fassten die Fragen, die die Teilnehmer während der Programmieraufgaben gestellt haben, zusammen und abstrahierten bzw. verallgemeinerten sie zu 20 Arten von Fragen. Hier einige interessante Fragen:

- „Hat ein API-Typ (z.B. eine Klasse) eine Methode, mit der ich eine bestimmte Aufgabe lösen kann? “
- „Wie kann ich ein Objekt eines bestimmten API-Typs (Klasse) ohne einen öffentlichen Konstruktor (public constructor) erstellen? “
- „Was ist der Zusammenhang zwischen den Typen (Klassen) X und Y?“

Wird die gesuchte Funktionalität in der API nicht einfach genug gefunden, sucht der API-Nutzer die Information im Internet. Das bedeutet, dass die

API-Dokumentation die Suche nach der richtigen Stelle möglichst einfach gestalten muss.
Die Studie deutet darauf hin, dass die Designentscheidung auf welche Art ein Objekt erstellt wird die API-Usability beeinflusst. Wird statt eines Konstruktors die Fabrikmethode verwendet, verwirrt das neue Nutzer einer API. Wenn für die Erfüllung einer Aufgabe mehrere Klassen gemeinsam genutzt werden müssen, muss die Beziehung der beiden Klassen zueinander geklärt werden. Diese Aufgabe wird von neuen API-Nutzern oft als schwierig empfunden. Dem kann Abhilfe geschafft werden, indem in der API-Dokumentation besonders darauf eingegangen wird oder möglichst simple API-Entwürfe verwendet werden.

Auflistung von Faktoren der API-Usability
Neben den vorgestellten Studien gibt es noch viele weitere, die unterschiedliche Faktoren finden, die die Benutzerfreundlichkeit einer API im positiven oder negativen Sinn beeinflussen([10,25–27] u.v.m.). Das führt zu einer sehr verteilten und unübersichtlichen Situation. Denn die Studien sind nicht alle einheitlich und deshalb ist es mühsam die in den Studien herausgefundenen API-Usability-Einflussfaktoren zusammenzutragen. Das wäre hilfreich, um einen Überblick über die API-Usability zu bekommen und, um Forschungslücken besser zu finden und zu füllen.
Zibran hat eine ausführliche Literaturstudie durchgeführt und eine Liste von Faktoren zusammengetragen, die die Benutzerfreundlichkeit von APIs beeinflussen [28]. Diese Liste (siehe Tabelle 1) wurde dann von Zibran et al. leicht überarbeitet [13]. Wie der Tabelle zu entnehmen ist, gibt es neben der API-Dokumentation weitere Faktoren, die die Usability einer API beeinflussen.

3.2 API-Dokumentation

In den Studien über API-Usability wird häufig die Relevanz von API-Dokumentationen verdeutlicht.
Aber selbst Software-Dokumentationen werden in vielen Fällen aus Kostengründen vernachlässigt [29], obwohl sie sehr wichtig für die Entwicklung und Wartung der Software sind [30]. Das macht es umso schwerer die Produktion und Pflege einer API-Dokumentation zu rechtfertigen [31]. Hierbei können Studien hilfreich sein, die die Wichtigkeit von API-Dokumentation für die API-Usability untersuchen und zeigen, dass es sich lohnt in eine gute API-Dokumentation zu investieren.

Einflussfaktor	**Beschreibung**
Komplexität	Die API sollte nicht zu viele und komplexe Funktionen, Konzepte und Architektur haben.
Benennung	Konsistente Benennung der API-Elemente anstreben, gängige Konventionen einhalten und deskriptive Namen nutzen
Nutzerperspektive	Aufruf und Nutzung der API-Elemente sollten klar und intuitiv sein
Dokumentation	Eine vollständige, klare und aktuelle Dokumentation mit Code-Beispielen und Use-Cases ist sehr wichtig
Konsistenz	Designentscheidungen sollten gängigen Konventionen entsprechen und über die ganze API hinweg konsistent bleiben
Konzeptuelle Korrektheit	Der Entwurf darf die Nutzung von zur Nutzung vorgesehenen Funktionalitäten der API nicht behindern
Parameter und Rückgabewerte	Zu viele Parameter oder unverständliche Rückgabewerte sind schlecht
Konstruktor und Parameter	Für Anfänger ist der parameterlose Konstruktor einfacher
Fabrikmethode vs. Konstruktor	Objekte sollten mit Konstruktoren und nicht mit der Fabrikmethode erstellt werden
Datentypen	Die angebotenen Objekte und Attribute sollten Typecasts unnötig machen
Attribute	Zu viele Attribute verringern die API-Usability
Nebenläufigkeit	thread-sichere Elemente anbieten
Fehlerbehandlung	Mechanismen für Fehlervorbeugung und informationshaltige Fehlermeldungen anbieten
Fehlende Funktionen	Funktionen, die viele Nutzer offensichtlich brauchen werden bereitstellen
Mehrere Wege zum selben Ziel	Es sollte möglichst einen Weg geben etwas zu erreichen
Aufrufkette	Lange Aufruf- und Hierarchieketten schaden der Übersichtlichkeit und somit der API-Usability
Unabhängigkeit	Die API sollte unabhängig von der Plattform oder anderen APIs funktionieren
Speicherverwaltung	Intern sollte eine effiziente Speicherverwaltung herrschen
Abwärtskompatibilität	Bei Updates sollte bereits geschriebener Code möglichst funktionieren
Aktualität	Die API sollte möglichst aktuell bleiben und neue Technologien unterstützen
Code-Lesbarkeit	Der Code der API sollte lesbar sein, falls mal in den Quellcode geschaut werden muss.

Tabelle 1: Die API-Usability betreffende Einflussfaktoren nach [13, 28]

API-Dokumentation ist wichtig für API-Usability

In der Studie von Zibran et al. [13] wurden die zuvor von Zibran [28] zusammengetragenen Faktoren, die die API-Usability beeinflussen, näher untersucht. Ziel war es, herauszufinden, wie signifikant die jeweiligen Faktoren für die API-Usability sind.

Methodik Um ihr Ziel zu erreichen, analysierten sie 1513 eingereichte Fehlerberichte (bug reports) aus den Projekten Eclipse[4], GNOME[5], MySQL[6], Python 3.1[7] und Android[8].
Sie betrachteten manuell alle 1513 Einträge und sortierten jene aus, die aus Sicht der API-Usability irrelevant erschienen und es blieben 562 Stück(∼37%) übrig.
Anschließend wurden diese näher überprüft, indem die Kommentare zu dem jeweiligen Bericht gelesen wurden. Dann wurden die Fehlerberichte in einem offenen Kodierverfahren den Einflussfaktor zugeordnet und dadurch konnte eine Statistik darüber erstellt werden, welche Art von Einflussfaktor wie oft in den Fehlerberichten auftauchen.

Ergebnisse Abbildung 2 zeigt die Verteilung der Fehlerberichte bezüglich ihrer Thematik. Etwa 43% der Fehlerberichte berichten eigentlich über keine Fehler, sondern fordern weitere Features. Danach kommen mit circa 32% „echte“ Fehlerberichte, also Fälle, in denen die API sich nicht so verhält, wie es von der Dokumentation versprochen wurde und wie es richtig sein müsste. Das beeinflusst aber mehr die Brauchbarkeit als die Benutzerfreundlichkeit. Denn bei einer unbrauchbaren API, indem Sinne, dass sie nicht funktioniert, erübrigt sich die Benutzerfreundlichkeit. Ungefähr ein Viertel (∼27%) melden Fehler in der Dokumentation. Neben vielen textuellen Fehlern (Tippfehler, grammatikalische Fehler) gibt es auch weitere Fehler, die auf eine falsche Beschreibung eines Elements oder auf falsche, nicht funktionierende Code-Beispiele basieren. In 22 der 117 Fehlerberichte, die von Dokumentationen handeln, wird angegeben, dass die Dokumentation eines oder mehrerer Elemente fehlen, diese also nicht beschrieben sind.
Grob jeder zehnte der analysierten Fehlerberichte dreht sich um die (Un-)Sichtbarkeit gewisser Elemente der API. In vielen Fällen fordern Nutzer der API Schnittstellen, über die sie auf interne Klassen oder Attribute zugreifen können.

[4] https://www.eclipse.org/ (letzter Zugriff: 10.05.2017)
[5] https://www.gnome.org/ (letzter Zugriff: 10.05.2017)
[6] https://www.mysql.com/ (letzter Zugriff: 10.05.2017)
[7] https://www.python.org/download/releases/3.1/ (letzter Zugriff: 10.05.2017)
[8] https://www.android.com/ (letzter Zugriff: 10.05.2017)

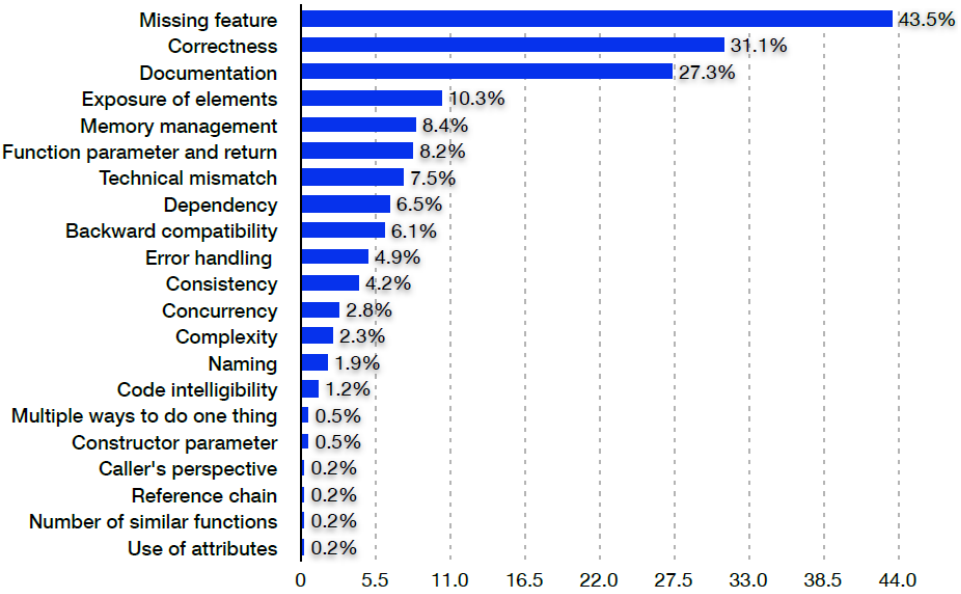


Abbildung 2: Verteilung der Fehlerberichte [Quelle: [13]]

Die Häufigkeit der restlichen Einflussfaktoren in den Fehlerberichten können Abbildung 2, Anhang A.1 oder der Studie [13] entnommen werden.
Sind solche Abhängigkeiten doch vorhanden, sollte das an mehreren Stellen verdeutlicht werden.

Kritik Die Datenerhebung bzw. -analyse fand vollständig manuell statt. Gerade bei der großen Anzahl der Daten und der Monotonie der Aufgabe ist das Risiko für menschliche Fehler groß. Es ist außerdem nicht ersichtlich, welcher von den Autoren wie viele Fehlerberichte zugeordnet hat und in wie fern es Kreuzkontrollen gab.
Als Motivation für die Studie wird angeführt, dass es wichtig wäre die jeweilige Signifikanz der die API-Usability beeinflussenden Faktoren besser zu kennen. Die Metrik die dafür genommen wurde, nämlich wie viele Fehlerberichte es zu den jeweiligen Faktoren gibt, ist aber nicht sehr geeignet. Wie die Autoren auch erkannten, kann nicht behauptet werden, dass die Speicherverwaltung für die API-Usability von größerer Bedeutung ist, als die Komplexität einer API, auch wenn es die Ergebnisse auf dem ersten Blick vermuten lassen.

Relevanz für diese Arbeit Die angewandte Methodik erlaubt keine absoluten Aussagen über die genaue Reihenfolge der Gewichtung der API-Usability beeinflussenden Faktoren. Dennoch liefert es aber Anzeichen dafür, dass die API-Dokumentation eines der wichtigsten API-Usability-Faktoren ist.

API-Dokumentation ist wichtig für API-Usability (II)
Grill et al. haben in ihrer Studie ein Verfahren aus der Forschung für Human-Computer-Interaction (HCI) mit einigen Anpassungen dazu verwendet die Benutzerfreundlichkeit einer API zu evaluieren und eventuelle Problemfelder aufzudecken [12]. Die HCI-Forschung ist ein interdisziplinäres Forschungsgebiet und beschäftigt sich damit, die Beziehung zwischen Technik und Mensch besser zu verstehen, um mit diesem Wissen bessere Technologien entwerfen zu könne [32].

Methodik Mit der Durchführung von drei unterschiedlichen Forschungsmethoden sollten unterschiedliche Probleme bzgl. der Benutzerfreundlichkeit der untersuchten API gefunden und besser verstanden werden. Diese Methoden sind: Heuristische Evaluation, Workshop, Interviews und wurden in dieser Reihenfolge durchgeführt.
Heuristische Evaluation: Zunächst wurden vier Experten, die sich mit der Domäne der genutzten API auskennen und erfahrene Programmierer sind, herangezogen und mit den Heuristiken vertraut gemacht. Die Heuristiken basieren auf den von Zibran zusammengetragenen API-Usability-Faktoren [13, 28] (siehe Tabelle 1). Die Experten sollten dann 110 Klassen und 29 Interfaces der API analysieren und gemäß den Heuristiken Probleme finden, die der Benutzerfreundlichkeit schaden. Die Probleme sollten ebenfalls in einer Skala von 1 bis 5 bewertet werden, wobei 1 bedeutet, dass das Problem harmlos ist, und 5 bedeutet, dass es ein sehr gravierendes Problem für die Usability ist.
Workshop: Nach der heuristischen Evaluation wurde ein Workshop mit acht Entwicklern durchgeführt. Zu Beginn des Workshops wurde den Teilnehmern die Domäne und die API, sowie die API-Dokumentation vorgestellt. Zusätzlich bekamen sie Fragebögen und einen Bogen, auf dem sie Probleme notieren sollten.
Nach der Einführung gab es detailliertere Tutorials bezüglich der API mit simplen Beispielen.
Danach bekamen die Teilnehmer Aufgaben, die mit dem Einsatz der API gelöst werden sollten. Die Aufgaben wurden so gestellt, dass möglichst alle Bereiche der API verwendet wurden. Während dieser Phase haben die Experimentatoren die Teilnehmer beobachtet und haben bei größeren Schwierigkeiten geholfen. Während des Workshops wurden für spätere Analysezwecke Ton- und Videoaufnahmen gemacht.
Interviews: Mit den Teilnehmern des Workshops wurden Interviews durchgeführt, in denen über die Erfahrungen im Workshop und über Erlernbarkeit, Einprägsamkeit, Effizienz, Missverständnisse, Fehler, Wahrnehmung der API, selbsterklärende Struktur, sowie Benennung gesprochen wurde.

Ergebnisse Im Folgenden werden nur die Ergebnisse vorgestellt, die etwas mit API-Dokumentation zu tun haben. Für eine ausführlichere Vorstellung der Ergebnisse, siehe A.2 oder Studie von Grill et al. [12]. Während der *heuristischen Evaluation* sind insgesamt 127 einzigartige Usability-Probleme ermittelt worden. Der durchschnittliche Schwierigkeitsgrad bzw. das durchschnittliche Ausmaß der Probleme betrug 3,1, wobei 1 für harmlos und 5 für folgenschwer steht. Die Kategorie, unter der die meisten Probleme fielen, ist die Dokumentation. 35 Mal wurde eine fehlende, falsche oder unvollständige Stelle in der Dokumentation gefunden.
Auch im *Workshop* gab es die meisten Usability-Probleme in der Dokumentation, nämlich 12 der insgesamt 44 im Workshop registrierten Usability-Probleme.
Dokumentation war auch im *Interview*-Teil das meist angesprochene Problemfeld.

Kritik Die Experimentatoren waren an allen Phasen der Studie beteiligt. Das bringt die Gefahr mit sich, dass sie die Teilnehmer der Studie beeinflusst haben. Obwohl der Workshop mit Kameras aufgenommen wurde, haben die Experimentatoren die Teilnehmer vor Ort beobachtet und schlimmer: Bei Problemen geholfen. Dabei sollten Probleme nicht nur identifiziert, sondern auch ihr Schwierigkeitsgrad bestimmt werden. Wenn einem die Lösung zu einem Problem gegeben wird, erscheint das Problem viel einfacher.
In dem Bericht über die Studie wird auf einige wichtige Punkte wenig eingegangen. die gewählten Heuristiken sind auf der in der Studie gelieferten Tabelle sehr kurz beschrieben. Haben die Experten, die diese Heuristiken benutzen sollten, detailliertere Beschreibungen bekommen?
Wie alltagsrelevant waren die Aufgaben, die die Teilnehmer des Workshops mithilfe der API lösen sollten? Das geht aus dem Bericht über die Studie nicht hervor, wäre aber wichtig, um die Ergebnisse entsprechend einordnen zu können. Hinzu kommt, dass nur eine API über einen kurzen Zeitraum beobachtet wurde. So sind die Ergebnisse leider wenig verallgemeinerbar.

Relevanz für diese Arbeit Das eigentliche Ziel der Studie war das Umändern und Ausprobieren von Usability-Evaluationsmethoden der HCI-Forschung, um die Usability einer API zu evaluieren. Das für diese Arbeit aber eigentlich Interessante an der Studie ist nur ein Nebenprodukt der Studie: Die Tatsache, dass die API-Dokumentation in allen drei Phasen am häufigsten als Usability-Problem identifiziert wurde.

Welche Rolle spielt API-Dokumentation für API-Usability?
Die Studien von Zibran et al. [13] und Grill et al. [12] liefern quantitative Hinweise dafür, dass die API-Dokumentation zur Zeit einer der negativsten Einflussfaktoren ist. Denn die meisten Probleme, die die API-Usability der

API verringern sind laut diesen Studien durch eine schlechte, fehlerhafte oder lückenhafte Dokumentation verursacht.
In der Studie von Uddin und Robillard geht es spezifisch um API-Dokumentationen [31]. Forschungsfrage war, wie sich zehn dokumentationsbezogene Probleme in der Praxis auswirken, welche am meisten vorkommen und wie schwerwiegend sie sind.

Methodik Zunächst wurde eine explorative Online-Umfrage mit fünf Fragen durchgeführt. Zweck dieser Umfrage war es, Erfahrungsberichte über gute und schlechte API-Dokumentationen zu erhalten. Zwei der fünf Fragen haben demographische Informationen gesammelt. Die restlichen drei Fragen haben nach der zuletzt genutzten API und dem Grund für die Nutzung dieser API gefragt, sowie nach je drei positiven und drei negativen Beispielen für API-Dokumentation mit einer jeweiligen Begründung.
69 der 698 angeschriebenen IBM-Mitarbeiter haben an der Umfrage teilgenommen. Die meisten (65) waren Software-Entwickler bzw. -Architekten.
Nach der ersten Umfrage gab es noch eine weitere Umfrage zur weiteren Vertiefung und Validierung der Ergebnisse aus der ersten Umfrage. In dieser Umfrage sollten die Teilnehmer ihre Erfahrung bezüglich der Probleme aus der ersten Umfrage wiedergeben. Für jedes Problem sollten die Teilnehmer angeben wie häufig sie diesem Problem begegnet sind und wie sehr sie davon abgehalten wurden ihre Aufgabe zu erfüllen. Den Schweregrad des Problems sollten sie mit einer Ordinalskala (Kein Problem, moderat, schwer, Blocker) bemessen, wobei Blocker bedeutet, dass das Problem nicht gelöst werden konnte und eine andere API genommen wurde. Außerdem sollten sie den Problemen Prioritäten zuordnen. Je höher die Priorität eines Problems, desto mehr Ressourcen würden die Teilnehmer der Lösung für dieses Problem bereitstellen. Es ist also ein Indikator dafür, wie wichtig sie eine Lösung halten und wie schwer sie die Lösung einschätzen.
Für diese Umfrage wurden 1064 Softwareentwickler der IBM Canada und der IBM Great Britain angeschrieben, von denen 254 antworteten. Teilnehmer der ersten Umfrage wurden nicht angeschrieben.

Ergebnisse Tabelle 2 zeigt wie oft welches Problem in den Antworten auf die erste Umfrage aufgetreten sind. Bei der Auswertung der Antworten wurde entschieden die Beispiele für positive Dokumentationen zu ignorieren, da diese laut Uddin und Robillard zu wenig Erkenntnisse gebracht haben. 61 von 86 genannten Problemen sind inhaltliche Probleme und die restlichen 25 sind Probleme, die die Präsentation der Dokumentation betreffen. Am meisten wurde eine unvollständige Dokumentation beanstandet. Der trivialste Fall hierfür sind automatisch generierte Dokumentationen, die nur die Signatur der Funktion zeigen und somit kaum eine Hilfe für den Nutzer der API darstellen. Bei API-Elementen, die neben einer komplexen Funk-

Kategorie	**Problem**	**Beschreibung**	Häufigkeit
Inhalt	Unvollständigkeit	Die Erklärung eines API-Elements bzw. eines Themas ist nicht vorhanden bzw. unvollständig.	20
	Unklarheit/Mehrdeutigkeit	Die Beschreibung eines API-Elements ist vorhanden aber unklar bzw. mehrdeutig.	16
	Unerklärte Beispiele	Die Erklärung zu Code-Beispielen fehlt oder ist unzureichend.	10
	Obsoleszenz	Eine Stelle in der API-Dokumentation beschreibt noch eine ältere Version.	6
	Inkonsistenz	Die Beschreibungen von Elementen, die zusammen genutzt werden sollen, widersprechen sich.	5
	Fehler	Fehlerhafte Beschreibung eines API-Elements bzw. eines Themas	4
		Gesamt	**61**
Präsentation \ Gestaltung	Informationsflut (Bloat)	Die Erklärung eines API-Elements bzw. eines Themas ist zu ausschweifend.	12
	Fragmentierung	Informationen über ein API-Element oder ein Thema sind stark verteilt.	5
	Exzessive Strukturinformationen	In der Beschreibung eines API-Elements wird zu sehr auf die Syntax oder Struktur eingegangen. Diese Information kann durch moderne IDEs viel leichter erlangt werden.	4
	verstrickte Information	Die Beschreibung eines API-Elements oder eines Themas beinhalteten Beschreibungen von anderen API-Elementen oder Themen, die der Nutzer nicht verlangt hat.	4
		Gesamt	**25**

Tabelle 2: Probleme in der API-Dokumentation, die in der explorativen Umfrage gefunden wurden.

tionalität sich auch auf andere API-Elemente beziehen, wird es schwer alle Einsatzzwecke des API-Elements abzudecken.
Das zweithäufigste Problem ist ebenfalls ein inhaltliches Problem und taucht auf, wenn die Beschreibung eines API-Elements unklar bzw. mehrdeutig ist. In solch einem Fall muss entweder durch ausprobieren, oder durch nachschauen im Quellcode, sofern zugreifbar, Klarheit geschaffen werden. Im schlimmsten Fall kann die Beschreibung falsch verstanden werden, was zu unerwartetem Programmverhalten führen und den API-Nutzer vor Schwierigkeiten stellen kann.
Zwölf Mal wurde das Problem der „Informationsflut“ (Bloat) genannt und ist somit das am dritthäufigsten genannte Problem. Wenn die Beschreibung eines API-Elements zu unstrukturiert, ausschweifend und länger ist als nötig, kann der lange Text den Leser abschrecken. Und wenn im Text viele Details vorhanden sind, die der Nutzer nicht wissen muss, die sehr offensichtlich sind oder, die er schon an mehreren Stellen gelesen hat, kann es den Leser verwirren und davon abhalten produktiv zu sein.
Das letzte Problem, das mehr als 10% der gemeldeten Probleme ausmacht, sind unerklärte Beispiele. In den zuvor vorgestellten Studien wurde die Wichtigkeit von Code-Beispielen angesprochen. Wenn diese Code-Beispiele keine Erklärung haben, können sie ihre Wirkung verfehlen und den Nutzer auf eine falsche Spur bringen.
Abbildung 3 zeigt die Ergebnisse der zweiten Umfrage. Die X-Achse zeigt wie häufig ein Problem genannt wurde und die Y-Achse steht für den Schwierigkeitsgrad des Problems. Die Größe der Kreise stellen die Priorisierung dar - je größer ein Kreis, desto höher die Priorität bzw. desto dringender sollten diese Art der Probleme gelöst werden. Auch hier dominieren die inhaltlichen Probleme. Interessanterweise hat das Problem der „Unvollständigkeit“ nicht die höchste Priorität bekommen, obwohl es am häufigsten auftritt und von den Teilnehmern als am schwierigsten genannt wurde. Die größte Priorität hat die „Unklarheit/Mehrdeutigkeit“ bekommen, was sowohl in der Häufigkeit, als auch im Schwierigkeitsgrad auf Platz 2 ist. Zwar viel seltener, aber von einem sehr ähnlichen Schwierigkeitsgrad sind „Fehler“ in der Dokumentation zu finden. Überraschend ist auch, dass „Informationsüberflut“ (Bloat) so selten und als kaum problematisch eingestuft wird. War es doch das am häufigsten genannte Problem in der ersten Studie. „Verstrickte Information“ und „Exzessive Strukturinformationen“ fehlen, weil ihnen keine Priorität zugewiesen wurde.
Die Ergebnisse zusammengefasst:

- Die inhaltlichen bzw. qualitativen Probleme der API-Dokumentationen beeinträchtigen die Benutzerfreundlichkeit der API-Dokumentation und somit auch der API am meisten.
- Die schwersten Probleme sind die, deren Lösung viel technisches Wissen und einen tiefen Einblick in die API erfordern

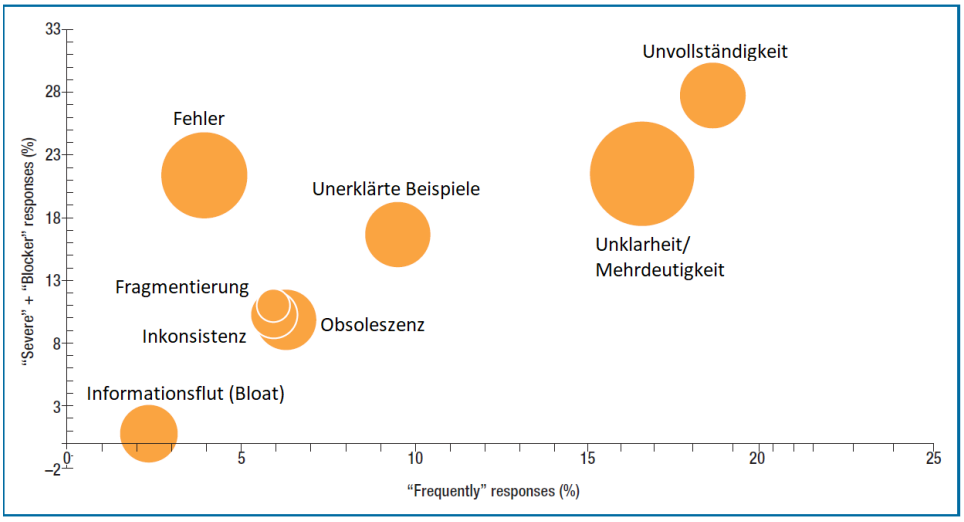


Abbildung 3: Häufigkeit und Gewicht der API-Usability-Probleme [Quelle: [31]]

Diejenigen, die diese Probleme am ehesten lösen können, sind aber entweder überfordert, oder arbeiten an etwas Anderem [31]. Dadurch wird die Dokumentation von Entwicklern mit wenig Erfahrung oder wenig Wissen über die API geschrieben, was zu diesen Problemen führt. Uddin und Robillard schließen daraus, dass die Entwickler, die diese Probleme lösen können mehr Unterstützung brauchen, damit sie mehr Zeit dafür haben diese Probleme zu lösen bzw. sie zu vermeiden.

Kritik Der Titel des Papers, indem die Studie veröffentlicht wurde, heißt „How API Documentation Fails“ (Deutsch: Wie API-Dokumentation fehlschlägt). Das erweckt zunächst den Eindruck, dass die Studie konkrete Informationen darüber liefert, *wie* API-Dokumentationen fehlschlagen. Stattdessen wird die Häufigkeit und der Schwierigkeitsgrad von Arten von API-Usability-Problemen vorgestellt, was zwar ebenfalls ein guter Beitrag ist, aber nicht so viel darüber aussagt, wie API-Dokumentationen nun versagen. Das ist aber Kritik auf einem sehr hohen Niveau und immerhin wird aus dem Titel ersichtlich, dass es sich um eine Studie handelt, welche die Probleme von API-Dokumentationen untersucht.
Eine Gefahr für die Gültigkeit der Ergebnisse ist die Frage, ob und wie die Teilnehmer der zweiten Umfrage über die zehn Problemklassen informiert wurden. Die Beschreibungen aus dem Paper (siehe Tabelle 2) sind sehr knapp und das Problem „Exzessive Strukturinformationen“ ist sehr ungenau bzw. subjektiv. Das könnte ein Grund dafür sein, weshalb die Teilnehmer so wenig drauf eingegangen sind. Wurden die Problemkategorien näher erläutert? Da es sich um eine E-Mail-Umfrage handelt und diese nicht zu viel

Text enthalten sollten, damit die potenziellen Teilnehmer nicht abgeschreckt werden, gehe ich nicht davon aus.

Relevanz für diese Arbeit Die Studie geht etwas genauer auf API-Dokumentation ein und liefert starke Hinweise dafür, dass es Verbesserungsbedarf gibt. Es wird von Problemen in der API-Dokumentation berichtet, die dazu führen, dass der Entwickler vor ein unlösbares Problem kommt und die API wechselt. Somit wird erneut die Notwendigkeit sichtbarer gemacht, dass API-Dokumentationen wichtig sind und verbessert werden müssen.

3.3 Wissenstypen in API-Dokumentationen

Welche Arten von Informationen gibt es in API-Dokumentationen?

In der Studie von Maalej und Robillard wird eine Taxonomie der Wissenstypen in API-Dokumentationen vorgestellt [1]. Mithilfe dieser Wissenstypen können API-Dokumentationen im Bezug auf den Inhalt der Beschreibungen analysiert werden. Durch die Analyse kann herausgefunden werden welches Wissen sie enthalten und wie dieses Wissen innerhalb der Dokumentation organisiert ist. Außerdem können die in der Taxonomie definierten Begriffe verwendet werden, um sich z.B. in Forschungen über API-Dokumentation besser verständigen zu können. Konkret lassen sich mit dieser Taxonomie die Dokumentationen unterschiedlicher APIs besser miteinander vergleichen.

Methodik Die Studie war in vier Phasen gegliedert.
Phase 1: Entwicklung der Taxonomie
Ziel dieser Phase war es eine Taxonomie für die Inhalte der API-Dokumentationen aufzustellen. Die Taxonomie sollte verlässlich und eindeutig sein, sodass verschiedene Leute der selben Dokumentationseinheit den selben Wissenstyp zuordnen. Außerdem sollte es alle für die praktische Softwareentwicklung relevanten Wissenstypen abdecken. Und anders als vorherige Taxonomien sollte es detaillierte und nicht grobe Eigenschaften auflisten.
Es wurde ein datengestützter Ansatz (englisch: grounded approach [33]) gewählt, um eine möglichst repräsentative Stichprobe an Dokumentationseinheiten von zwei APIs zu ziehen (HttpComponents[9] und Jena[10]). Da API-Dokumentationen in natürlicher Sprache geschrieben sind, ist es schwer sie zu analysieren. Deswegen fiel es den Autoren schwer auf Anhieb Wissenstypen zu extrahieren. Sie haben sich Fragen überlegt, die ein Satz bzw. eine Stelle der Dokumentation möglicherweise beantworten kann. Dann haben sie diese Fragen nach und nach zusammengefasst und sind auf zwölf Informationstypen samt Beschreibung gekommen. Die Beschreibungen wurden im Laufe der Zeit und der nächsten Phasen der Studie weiter verfeinert.

[9] http://hc.apache.org/(letzter Zugriff: 10.05.2017)
[10] http://jena.apache.org/(letzter Zugriff: 10.05.2017)

Phase 2: Anwendung der Taxonomie
Ziel dieser Phase war es zu zeigen, dass die Taxonomie funktioniert und die API-Dokumentationen der Java JDK 6.0 und der .NET 4.0-Plattform zu typisieren, also sie bezüglich der Wissenstypen zu analysieren. Auch hier wurde wieder eine Stichprobe aus den Dokumentationen beider APIs erstellt und dabei die geschichtete Zufallsauswahl [33] verwendet, damit eine möglichst fair zusammengestellte Stichprobe entsteht. Die Stichprobe bestand aus 5574 Dokumentationseinheiten. von jedem dieser Dokumentationseinheiten sollten die Wissenstypen von zwei unterschiedlichen Kodierern kodiert werden und das bedeutete 11148 Kodierungen. 17 Kodierer wurden mit der Taxonomie ausgiebig vertraut gemacht. Sie bekamen ein von den Studienführern entwickeltes Werkzeug namens „Content Analysis for API Documentation tool“ (CADo)[11]. Dieses Programm zeigt dem Nutzer, also dem Kodierer, eine Dokumentationseinheit. Außerdem gibt es zwölf Checkboxen für die Wissenstypen, die vom Kodierer angekreuzt werden müssen, wenn der entsprechende Wissenstyp in der Dokumentationseinheit vorhanden ist.
Phase 3: Überprüfung der Ergebnisse
In dieser Phase sollte gezeigt werden, dass die Methode, die bei der Anwendung der Taxonomie, gültige Ergebnisse geliefert hat. Dazu wurde die Übereinstimmungsquote der Kodierer untersucht. Wenn die Kodierer die Dokumentationseinheiten oft unterschiedlich bewertet hätten, wäre das ein Hinweis dafür, dass die Taxonomie mehrdeutig und deswegen den eigenen Ansprüchen widersprechen würde. Außerdem wurden Unstimmigkeiten in der Bewertung beseitigt.
Phase 4: Analyse der Ergebnisse
In dieser Phase wurden die in der zweiten und dritten Phase gesammelten Ergebnisse ausgewertet. Dazu wurden Statistiken erstellt über die Häufigkeit der Wissenstypen. Außerdem wurden Beziehungen zwischen den Wissenstypen betrachtet, also, ob manche Wissenstypen besonders oft bzw. sehr selten gemeinsam auftreten. Des Weiteren wurde der Einfluss der Länge einer Dokumentationseinheit betrachtet, welche Wissenstypen enthalten sind und welche nicht.

Ergebnisse Die herausgearbeitete Taxonomie für API-Dokumentationen enthalten zwölf Wissenstypen. Ein Wissenstyp sagt aus, welcher Art von Informationen eine Stelle in der API-Dokumentation hat. In der Tabelle 3 sind die Wissenstypen in zusammengefasster Form beschrieben. Um eine Verfälschung zu vermeiden und Vergleiche zu anderen Studien, die Wissenstypen verwenden, zu vereinfachen, wurde auf eine Übersetzung verzichtet.
Unter den Kodierern gab es eine Übereinstimmungsquote von 82%, was angesichts der Komplexität der Aufgabe hoch ist. Abbildung 4 zeigt den durchschnittlichen Anteil von Wissenstypen in Dokumentationseinheiten aus der

[11] https://cado.informatik.uni-hamburg.de/tool/ (letzter Zugriff: 10.05.2017)

Knowledge Type	**Description (Excerpt)**
Functionality and Behaviour	Describes what the API does (or does not do) in terms of functionality or features. Describes what happens when the API is used (a field value is set, or a method is called).
Concepts	Explains the meaning of terms used to name or describe an API element, or describes design or domain concepts used or implemented by the API.
Directives	Specifies what users are allowed / not allowed to do with the API element. Directives are contracts.
Purpose and Rationale	Explains the purpose of providing an element or the rationale of a certain design decision. Typically, this is information that answers a "why" question: Why is this element provided by the API? Why is this designed this way? Why would we want to use this?
Quality Attributes and Internal Aspects	Describes quality attributes of the API, also known as non-functional requirements, for example, the performance implications. Also applies to information about the API's internal implementation that is only indirectly related to its observable behavior.
Control-Flow	Describes how the API (or the framework) manages the flow of control, for example by stating what events cause a certain callback to be triggered, or by listing the order in which API methods will be automatically called by the framework itself.
Structure	Describes the internal organization of a compound element (e.g. important classes, fields, or methods), information about type hierarchies, or how elements are related to each other.
Patterns	Describes how to accomplish specific outcomes with the API, for example, how to implement a certain scenario, how the behavior of an element can be customized, etc.
Code Examples	Provides code examples of how to use and combine elements to implement certain functionality or design outcomes
Environment	Describes aspects related to the environment in which the API is used, but not the API directly, e.g., compatibility issues, differences between versions, or licensing information.
References	Includes and pointer to external documents, either in the form of hyperlinks, tagged "see also " reference, or mentions of other documents (such as standards or manuals).
Non-information	A section of documentation containing any complete sentence or self-contained fragment of text that provides only uninformative boilerplate text.

Tabelle 3: Die Wissenstypen in API-Dokumentation nach [1]

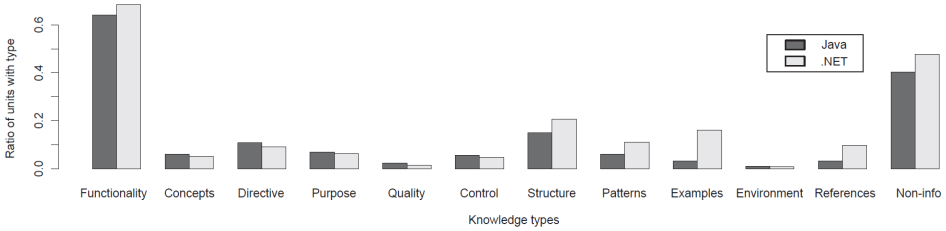


Abbildung 4: Verteilung der Wissenstypen in den Dokumentationseinheiten in JDK 6 und .NET 4.0 [Quelle: [1]]

API-Dokumentation von JDK 6 bzw. .NET 4.0 auftauchen. Der am häufigsten vertretene Wissenstyp ist „Functionality and Behavior“, was nicht verwunderlich ist, da es ja die Hauptaufgabe von API-Dokumentationen ist, die Funktionalität und das Verhalten der API und ihrer Elemente zu beschreiben. Etwas überraschend erscheint der hohe Anteil an „Non-Information“. Das ist vermutlich der Form der API-Dokumentationen beider Plattformen geschuldet, die für jedes Element eine eigene Beschreibung vorsieht, egal wie trivial sie ist, oder, ob sie woanders schon beschrieben wurde. Allgemein lässt sich sagen, dass die Verteilung sehr ähnlich ist. Eine Ausnahme bilden die Wissenstypen „Code-Examples“ und „References“.
Was die Korrelation von Wissenstypen betrifft, ist die Tatsache, dass „Patterns“ häufig mit „Code-Examples“ korrelieren, nicht überraschend, da es in der Natur der Wissenstypen liegt, dass Code-Beispielen eine Erklärung beiliegt.
„Structure“ und „Patterns“ korrelieren ebenfalls stark, da vermutlich in einer Beschreibung, in der erklärt wird, wie etwas zu erreichen sind, auf die unterschiedlichen Wissenstypen verwiesen wird.
Die Längenanalyse hat wie erwartet ergeben, dass je länger eine Dokumentationseinheit ist, desto mehr unterschiedliche Wissenstypen enthalten sind.

Kritik In der Studie wurde lediglich das Vorhandensein von bestimmten Wissenstypen in Dokumentationseinheiten gemessen. Das ist eine ungenaue Metrik, denn sie sagt nichts über die Längen der Wissenstypen in diesen Dokumentationseinheiten aus. Aus den Ergebnissen dieser Studie könnte z.B. der falsche Schluss gefolgert werden, dass es in den API-Dokumentationen von Java bzw. .NET mehr redundante Informationen enthalten sind, als Informationen über die Struktur. Es kann aber sein, dass die Struktur zwar in weniger Dokumentationseinheiten auftauchen, dafür aber viel mehr beschrieben werden, als „Non-Information“. Um noch genauere Aussagen über den Gebrauch von Wissenstypen in API-Dokumentationen sprechen zu können, müsste nicht nur geschaut werden, *ob* ein Wissenstyp vorhanden ist, sondern wie viel Text für einen Wissenstyp verwendet wird.

Relevanz für diese Arbeit Die Studie von Maalej und Robillard ist sehr wichtig für diese Arbeit, da die Wissenstypen auch in dieser Arbeit eine Rolle spielen und die Idee für diese Arbeit zum Teil von deren Studie inspiriert wurde.

Welche Unterschiede gibt es zwischen der Java- und der Python-API bzgl. der Wissenstypen?
In der Bachelorarbeit von Wildermann wurde die Studie von Maalej und Robillard [1] (siehe Tabelle 3) auf die Dokumentation der Python Standardbibliothek mit einigen Änderungen erweitert. Die wichtigste Änderung ist, dass nicht mehr anhand von Checkboxen markiert wird, ob ein Wissenstyp in einer Dokumentationseinheit vorkommt. Stattdessen wird markiert welcher Textabschnitt welchen Wissenstypen enthält. Das erlaubt genauere Aussagen über die Verteilung der Wissenstypen in API-Dokumentationen.

Methodik Es wurde die Taxonomie von Maalej und Robillard [1] übernommen und das dazugehörige Kodierhandbuch, das von Kodierern dazu verwendet wird unterschiedliche Wissenstypen voneinander unterscheiden zu können, wurde mit Informationen bzw. Anweisungen zu Eigenheiten der Dokumentation von der Python Standardbibliothek(Python-API-Dokumentation) ergänzt.
Außerdem wurden an dem Kodierhandbuch Änderungen vorgenommen[12], die die veränderte Vorgehensweise widerspiegeln. Diese Änderungen erklären, dass der Text markiert werden soll, der zu einem bestimmten Wissenstyp passt. Vorher hieß es, dass die entsprechenden Checkboxen markiert werden sollen.
Außerdem wurde eine Software geschrieben, die von Kodierern genutzt werden sollte. Diese Software gab jedem Kodierer eine Dokumentationseinheit und die Möglichkeit den Text mit dem Mauszeiger in entsprechenden Farben zu markieren, wobei jede Farbe für einen Wissenstyp stand.
Es wurde eine sogenannte Goldstichprobe erstellt, die aus 168 Dokumentationseinheiten bestand. Diese Stichprobe wurden von drei Mitgliedern der AG Software Engineering der Freien Universität Berlin und dem Autor der Bachelorarbeit typisiert. Das sollte dazu dienen einen Referenzwert für hohe Typisierungsqualität zu liefern.
Für die Hauptstichprobe, die aus 1548 Dokumentationseinheiten bestand, wurde ein Forschungspraktikum an der Freien Universität Berlin veranstaltet. In diesem Forschungspraktikum wurden sieben Studenten mit der Taxonomie, dem Kodierhandbuch und der Software vertraut gemacht, die sie für die Typisierung nutzen sollten.

[12] http://www.inf.fu-berlin.de/w/SE/ThesisPythonDokuInfotypenCodingHandbook (letzter Zugriff: 10.05.2017)

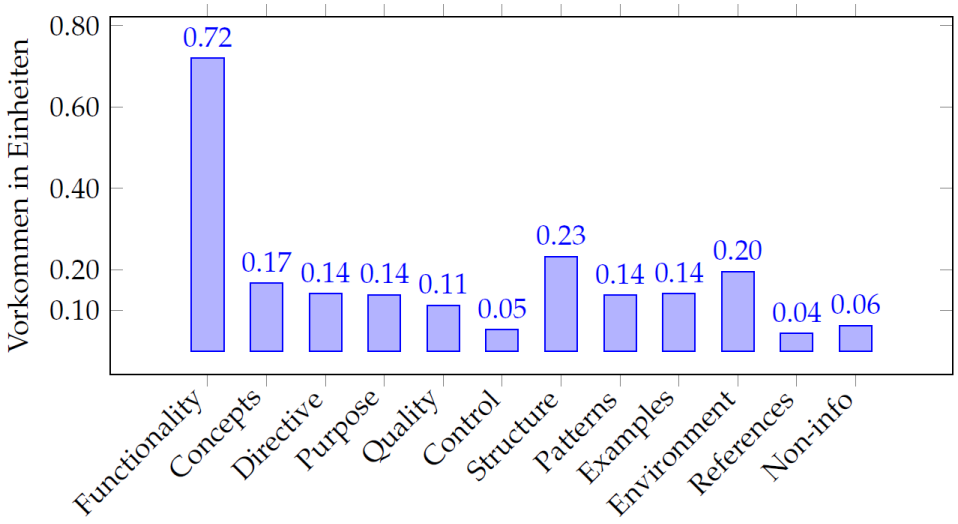


Abbildung 5: Verteilung der Wissenstypen in Dokumentationseinheiten der Python Standardbibliothek [Quelle: [34]]

Nach der Typisierung wurde ähnlich der Originalstudie die Übereinstimmung unter den Kodierern überprüft, Unstimmigkeiten gelöst und die Ergebnisse analysiert.

Ergebnisse Die durchschnittliche Übereinstimmung der Kodierungen unter den Entwicklern betrug 87,4%. Das heißt aber noch nicht, dass die Entwickler korrekt typisiert haben. Sie können ja alle gleichviel falsch typisiert haben. Ein Vergleich der Typisierung der Hauptstichprobe zur Goldstichprobe ergab eine Übereinstimmung von durchschnittlich 92,2%. Das bedeutet, dass die Typisierung der Kodierer sehr gut war.
Abbildung 5 zeigt die Verteilung der Wissenstypen in den Dokumentationseinheiten Python-API-Dokumentation. Der größte Unterschied zu den Wissenstypen aus den Dokumentationen der Java- und .Net 4.0-API betrifft den Wissenstypen „Non-Information“, der in Python viel weniger seltener ist. Das bekräftigt meinen Verdacht, den ich bei der Vorstellung der Studie von Maalej und Robillard [1] geäußert habe. Im Gegensatz zu den Dokumentationen der Java- und .Net 4.0-APIs, gibt die Dokumentation der Python Standardbibliothek keine starre Vorgabe, die vorsieht, dass alle Elemente nochmal einzeln erklärt werden müssen. Diese Beobachtung wird in Kapitel 5 nochmal wichtig. Eine weitere Auffälligkeit ist der Unterschied beim Wissenstyp „Environment“, der bei Python viel höher ist. Das deutet auf Platformabhängigkeit hin, was für eine API eigentlich eher unerwünscht ist. Die restlichen Wissenstypen sind mehr oder weniger gleichermaßen vorhanden.
Was die Korrelation der Wissenstypen angeht, ist die Kombination „Pat-

terns“ und „Code-Examples“ wieder sehr hoch. „Structure“ und „Concepts“ tauchen ebenfalls oft zusammen auf. Das liegt nahe, dass bei Erklärung von Konzepten auch drauf eingegangen wird, welche Elemente damit etwas zu tun haben.
Auch in der Bachelorarbeit von Wildermann liefert die Längenanalyse keine spannenden Ergebnisse. Je länger eine Dokumentationseinheit bzw. eine Dokumentationsseite ist, desto mehr unterschiedliche Wissenstypen sind enthalten. Dabei sind Modulbeschreibungen, die meistens eine ganze Dokumentationsseite ausmachen, am längsten, gefolgt von Klassenbeschreibungen, Methodenbeschreibungen, Feldbeschreibungen und sonstigen Beschreibungen.

Kritik Bei den Auswertungen der Ergebnisse wird leider überhaupt nicht auf die Längen der jeweiligen Wissenstypen eingegangen. Dabei wurde die Kodierweise im Gegensatz zur Originalstudie so geändert, dass Aussagen über die Längen der Wissenstypen möglich wären. Die Änderung hat die Typisierung wesentlich aufwändiger gemacht. Der potenzielle Vorteil der sich daraus ergibt, wurde nicht genutzt. Da in der Bachelorarbeit von Wildermann dieser Aspekt öfter erwähnt wurde, gehe ich davon aus, dass dafür keine Zeit mehr übrig blieb.

4 Forschungsmethoden

In diesem Kapitel werden die Forschungsmethoden beschrieben, die während der Forschung für die Masterarbeit eingesetzt wurden. Da ich überwiegend explorative und qualitative Forschung betrieben habe, war es nicht möglich die einzusetzenden Forschungsmethoden im Voraus zu planen und einfach nacheinander durchzuführen. Das liegt zum einen daran, dass zu Beginn der Masterarbeit nicht feststand, welcher Theorie überhaupt nachgegangen werden sollte. Am Anfang gab es nur die grobe Vermutung, dass die Dokumentationen der Java^{TM} Platform, Standard Edition 6[13] (Java-API-Dokumentation) und der Dokumentation der Python Standard Library[14] (Python-API-Dokumentation) unterschiedliche Stile haben. Und da beide Dokumentationen ganz gut sind, so die Annahme, lag der Gedanke nahe, dass es mindestens zwei Dokumentationsstile gibt, die unterschiedliche Vor- und Nachteile haben, und es sich lohnen könnte diese Stile zu identifizieren und ihre Eigenschaften und die damit verbundenen Vor- und Nachteile vorzustellen.
Allerdings war erstens nicht klar, inwiefern das untersuchbar ist und zweitens, ob das ein geeignetes Thema für eine Masterarbeit ist. Da wenig Forschung über API-Dokumentationen betrieben wurde, gibt es infolgedessen wenig Literatur über dieses Thema (siehe Kapitel 3). Um der These nachzugehen, gab es zu Beginn nur einen Anhaltspunkt: die Wissenstypen von Maalej und Robillard [1]. Mit der Zeit habe ich ein Gefühl für den Aufbau und die Struktur der beobachteten API-Dokumentationen entwickelt und ein besseres Verständnis für die Problemstellung bekommen. Dadurch konnte ich eigene Methoden entwickeln diese zu untersuchen. Zu diesem Zweck habe ich mich bereits vorhandener Tools bzw. Programme bedient und als es nötig wurde, mir kleine Java-Programme geschrieben, die mir Statistiken geliefert haben, z.B. über die Längen der API-Dokumentationen oder an wie vielen verschiedenen Klassen die Autoren geschrieben haben.

4.1 Typisierung der Dokumentationen

Auch wenn ich mich im Laufe meines Studiums mit der Dokumentation der Python-API und sehr viel mit der Dokumentation der Java-API beschäftigt habe, wusste ich sehr wenig darüber, wie sie inhaltlich aufgebaut sind. Denn ich war eher damit beschäftigt mein Programmierproblem zu lösen, als mich mit der API-Dokumentation selbst zu beschäftigen. Aus diesem Grund musste ich mich erstmal in diese API-Dokumentationen einarbeiten. Es war sehr wichtig zu verstehen, welche Typen von Informationen in diesen Dokumentationen enthalten sind, wie diese Informationen vermittelt werden

[13] http://docs.oracle.com/javase/6/docs/api/index.html (letzter Zugriff: 10.05.2017)
[14] https://docs.python.org/3.4/library/ (letzter Zugriff: 10.05.2017)

und wie die Dokumentation strukturiert ist. Hierfür waren die von Maalej und Robillard entwickelten [1] Wissenstypen und das von Wildermann um Python erweiterte Kodierhandbuch [34] sehr hilfreich.
Anders als die in den jeweiligen Studien [1,34] teilgenommene Kodierer wurde ich nicht von den Autoren in die Wissenstypen eingewiesen. Ich hatte aber zum einen das Kodierhandbuch, in dem die Wissenstypen ausführlich erklärt werden, und zum anderen die Typisierung aus beiden Studien. Mit diesen Daten konnte ich gut üben, indem ich zuerst die Dokumentationseinheiten der Goldstichprobe selbst typisiert habe und sie dann mit der Typisierung aus der Goldstichprobe verglichen habe. So konnte ich meine Missverständnisse bezüglich der Wissenstypen beseitigen.
Mit den Typisierungen aus Maalej und Robillards Studie und Wildermanns Bachelorarbeit kann ein quantitativer Vergleich zwischen den Dokumentationen der Java- und der Python-API gezogen werden, indem die Verteilung der in den Dokumentationseinheiten vorkommenden Wissenstypen verglichen wird. Der Vergleich hat aber keine gravierenden Entdeckungen bezüglich der Dokumentationsstile hervorgebracht. Das liegt zum einen daran, dass die reine Information darüber, ob ein bestimmter Wissenstyp in einer Dokumentationseinheit enthalten ist, eine sehr schwache Aussagekraft hat. Es könnte nämlich sein, dass sie zwar viele Informationstypen enthält, aber die meisten nur jeweils ganz kurz in einem Satz vorkommen und der Rest des Textes nur einen bestimmten Wissenstyp enthält. Besser wäre es also nicht nur zu wissen, ob ein Wissenstyp enthalten ist, sondern welchen Anteil am Text er ausmacht. Das ließe sicherere, genauere und nützlichere Aussagen über mögliche Stile treffen.
Daher war die Verwendung der Methodik von Wildermann, mit der eben solche Informationen über die Dokumentationseinheiten der Python-Doku erhalten werden können, ein möglicher Ansatz, um vergleichbare Daten zu bekommen.
Jedoch ist es schwierig nur mit quantitativen Ergebnissen Stile zu beschreiben und zu untersuchen. Qualitative Ergebnisse, wie z.B. die Reihenfolge, in der die Wissenstypen vorkommen, können Erkenntnisse über die Strategie liefern, die in einer API-Dokumentation verfolgt wird, um Wissen über die API zu vermitteln. Es wäre spannender nicht nur die Reihenfolgen innerhalb der Dokumentationseinheiten zu betrachten, sondern auch, ob es bestimmte häufige Abfolgen von Wissenstypen in einer Dokumentationsseite gibt. Das könnte besonders in der Dokumentation der Python-API interessant sein, da dort keine sofort erkennbare, einheitliche Struktur unter den Dokumentationsseiten erkannt werden kann, anders als es bei der Dokumentation der Java-API ist.
Aus diesem Grund und weil der Aufwand, die Methodik von Wildermann nochmal mit der Dokumentation der Java-API durchzuführen, sehr groß und der Nutzen nicht absehbar ist, habe ich mich für einen anderen ersten Ansatz entschieden.

Um beide Dokumentationen möglichst fair vergleichen zu können, wollte ich die Dokumentationen von Klassen und Modulen bzw. Objekten betrachten, die dieselbe Funktionalität erfüllen. Die Idee war, zu beobachten, wie auf beiden Plattformen die gleichen Funktionalitäten beschrieben sind. Denn, wenn sie eine ähnliche Strategie bzw. einen ähnlichen Stil hätten, müsste sich auch die Dokumentation ähneln. Erwartet wurde aber eine unterschiedliche Dokumentation, trotz gleicher Funktionalität. Es sollten Dokumentationen von API-Elementen mit ähnlichen Funktionalitäten sein, um nicht den sprichwörtlichen Vergleich zwischen Äpfeln und Birnen zu ziehen. Die Unterschiede in der Art und Weise, wie das Gleiche beschrieben und erklärt wurde, sollten betrachtet werden und daraus unterschiedliche Strategien, Eigenschaften und Stile herausgefunden werden.
Die Suche nach solchen Pendants hat sich als schwierig herausgestellt und die Erfolgsaussichten waren gering. Nicht nur, weil es auf beiden Seiten eine sehr große Anzahl an Dokumentationsseiten gibt[15], sondern auch weil die Struktur bzw. der Entwurf bei beiden APIs unterschiedlich ist. Ein weiterer Unterschied ist die Typisierung der beiden Sprachen. Java ist statisch und strenger typisiert, als Python. Außerdem können bei Python beim Aufruf einer Methode mit Parametern auf die Parameter verzichtet werden. In diesem Fall wird der Standardwert der Parameter übernommen. Das sorgt für unterschiedliche Entwurfsmöglichkeiten der APIs. Hinzu kommt, dass es bei den API-Dokumentationen ebenfalls unterschiedliche Designentscheidungen gibt. So haben Klassen und Interfaces in Java ihre eigenen Dokumentationsseiten. In Python hingegen wird meistens ein ganzes Modul in einer Dokumentationsseite beschrieben und enthält somit die Dokumentationen mehrerer Klassen.
Ich vermutete, dass das datetime-Modul[16] aus der Python-API und die Klasse Date[17] aus der Java-API gut vergleichbar wären. Doch erstens sind die meisten Elemente aus der Date-Klasse deprecated[18] und zweitens gibt es viele Funktionalitäten im datetime-Modul, wie z.B. die Möglichkeit zwischen Zeitzonen zu konvertieren, die es in der Klasse Date in der Java-API nicht gibt.
Besonders dem zweiten Problem bin ich sehr oft begegnet. Ich habe Module bzw. Klassen gesucht, die allgemeine Probleme lösen, bei denen ich die Vermutung hatte, dass es nicht viele unterschiedliche Lösungsmöglichkeiten gibt. Das sind z.B. Module bzw. Klassen, bei denen die Konzepte extern gegeben sind, wie z.B. Zeit oder Mathematik. Oder aber Module bzw. Klassen, die technische Sachen umsetzen, bei denen die Spezifikation vorgegeben

[15] ca. 3793 bei der Java-API und ca. 285 bei der Python-API
[16] https://docs.python.org/3.6/library/datetime.html (letzter Zugriff: 10.05.2017)
[17] https://docs.oracle.com/javase/6/docs/api/java/util/Date.html (letzter Zugriff: 10.05.2017)
[18] veraltet

ist. Daher habe ich mir ebenfalls die für die Erstellung und Nutzung von Sockets relevanten Module angeguckt. Aber auch da hat sich herausgestellt, dass das socket-Modul[19] und die Socket-Klasse[20] keine guten Pendants sind. Auch hier war das Problem, dass die Java-API eine spezifischere Unterteilung gewählt hat, was die Modularisierung betrifft, während die Python-API ein Objekt für Sockets hat, das entsprechend eingestellt werden kann, um verschiedenen Zwecken zu dienen, wohingegen die Java-API eigene Klassen für unterschiedliche Socket-Typen vorsieht.
Diese Probleme ließen sich teilweise dadurch überwinden, dass ich versucht habe ein Modul der Python-API mit unterschiedlichen Klassen und Methoden der Java-API nachzubilden und somit auf der Java-Seite ebenfalls ein Pendant zu erhalten. Dadurch entstanden Pendants mit möglichst ähnlichen Funktionalitäten. Das brachte allerdings den Nachteil mit sich, dass so keine durchgängigen Dokumentationsseiten mehr existierten. Außerdem war es schwierig eine „Klassenbeschreibung“ zusammenzustellen, die in den Dokumentationen der Java-API oft als Einleitung und Übersicht dienen. Ich konnte nicht einfach die gesamten Klassenbeschreibungen aller Klassen nehmen, aus denen ich Elemente für die Java-Seite genommen habe. Nicht nur, weil es zu redundanten Informationen geführt hätte, weil in mehreren Klassenbeschreibungen das gleiche Konzept erklärt wird, sondern auch, weil manche Klassenbeschreibungen Informationen zu API-Elementen aus der Klasse enthalten hätten, die ich nicht in die Java-Seite aufgenommen habe. Wenn ich also alle Klassenbeschreibungen übernommen hätte, hätte das zu einer sehr langen Klassenbeschreibung der von mir zusammengestellten Klasse bzw. Dokumentationsseite geführt, in der manche Informationen mehrmals enthalten wären und manche Informationen für die Klasse komplett irrelevant wären. Es war also wichtig, für die Klassenbeschreibung des erstellten Pendants auf der Java-Seite nur relevante Abschnitte aus den jeweiligen Klassenbeschreibungen zu übernehmen.
Ein weiteres Problem, das diese Vorgehensweise mit sich brachte, war, dass manche Methoden auf der Python-Seite nur mit einer Kombination von mehreren (meist zwei) Methoden auf der Java-Seite realisierbar waren.
Trotz dieser Schwierigkeiten erschien mir die Vorgehensweise legitim. Wie Abbildung 6 zeigt, habe ich die Beschreibungen der zueinander gehörenden Methoden aus dem entsprechenden Python-Modul und der Java-Klasse in einem Word-Dokument untereinander platziert. Ein horizontaler Strich zwischen den beiden Beschreibungen und ein Rahmen um beide Beschreibungen haben für eine bessere Übersichtlichkeit gesorgt. Ich habe für jeden Wissenstyp eine Farbkombination für Schrift und Schrifthintergrund bestimmt und die Beschreibungen typisiert. Das Gleiche habe ich für die Klassen- und

[19] https://docs.python.org/3.6/library/socket.html (letzter Zugriff: 10.05.2017)

[20] http://docs.oracle.com/javase/6/docs/api/java/net/Socket.html (letzter Zugriff: 10.05.2017)

socket.gettimeout()

Return the timeout in seconds (float) associated with socket operations, or None if no timeout is set. This reflects the last call to setblocking() or settimeout().

---

public int getSoTimeout()
throws SocketException

Returns setting for SO_TIMEOUT. 0 returns implies that the option is disabled (i.e., timeout of infinity).

Returns:
the setting for SO_TIMEOUT

Throws:
SocketException - if there is an error in the underlying protocol, such as a TCP error.

Since:
JDK1.1

See Also:
setSoTimeout(int)

Abbildung 6: Direkter Vergleich der Wissenstypen von zwei ähnlichen Methoden aus Python und aus Java

	H	I	J	K	L	M
48						
49	socket.gettimeout()	86		114	getSoTimeout()	
50		53		55		
51						
52	socket.listen([backlog])	32		205	listen(int backlog)	
53		37		39		
54		166				
55		53				

Abbildung 7: Untersuchung auf Abfolgen von Wissenstypen in den Dokumentationen der Java- und Python-API

Feldbeschreibungen ebenfalls getan.
Um häufig auftretende oder besonders interessante Abfolgen von Wissenstypen besser sehen zu können, habe ich die Informationen über die Wissenstypen in einer Excel-Tabelle zusammengetragen. Abbildung 7 zeigt einen Ausschnitt aus dieser Tabelle.
Die Spalten H und L enthalten jeweils den Namen des Elements um das es geht, wobei Spalte H die Elemente aus der Python-API und Spalte L die Elemente aus der Java-API enthält. Jede farbige Zelle in den Spalten I und K stehen jeweils für einen Textabschnitt mit einem Wissenstyp. Wieder ist die Kombination aus Schrift- und Hintergrundfarbe der Indikator für den Wissenstyp. Die Zahl in der Zelle gibt an, wie viele Zeichen der jeweilige Textabschnitt ohne Leerzeichen enthält.
Mit dieser Forschungsmethode sind mir auf beiden Seiten gewisse Muster

aufgefallen, die ich dann näher untersucht habe. Manche Muster, wie z.B. der Wissenstyp „Pattern“, der beschreibt, wie etwas mit der API umgesetzt werden kann, gefolgt vom Wissenstyp „Code Example“ waren wenig überraschend und nicht sehr relevant. Einige der in Kapitel 5 vorgestellten Stileigenschaften wurden mittels dieser Methode entdeckt.

4.2 Grounded Theory Methodology

Die Grounded Theory Methodology (GTM) ist eine überwiegend in den Sozialwissenschaften eingesetzte Methode. Sie wird bei explorativen, qualitativen Studien eingesetzt und soll dabei helfen aus einer Datensammlung Theorien zu bilden. Außerdem soll die GTM für die Forschung eines wenig erforschten Gebietes geeignet sein [35]. Meine Arbeit hat einige sozialwissenschaftliche Aspekte, weil es unter anderem um Didaktik, Verfassen von Texten und Lernstrategien geht, ist aber Forschung in der Informatik und somit eher naturwissenschaftlich. Bei meiner Studie handelt es sich jedoch um eine explorative, qualitative Studie, deren Forschungsgebiet wenig erforscht ist (siehe Kapitel 3). Des Weiteren haben sich Salinger und Kahlert in ihren Dissertationen [17, 36] ausführlich mit dem Thema auseinandergesetzt und gezeigt, dass die GTM auch für Studien in der Informatik geeignet sein kann.
Es gibt mehrere Formen der GTM. Im Folgenden werde ich die Prinzipien und Phasen der GTM nach Glaser, Strauss und Corbin [37, 38], die von Salinger und Kahlert zusammengefasst wurden [17, 36], kurz vorstellen.

Prinzipien

1. **Iteratives Vorgehen:** Die unterschiedlichen Phasen in der Forschung, wie z.B. Datenanalyse oder Theoriebildung, werden nicht sequenziell nacheinander bearbeitet, sondern, bevor eine Phase fertig ist, wird mit der nächsten schon begonnen und es wird immer wieder in eine vorherige Phase zurückgekehrt und dort fortgefahren.

2. **Theoretisches Sampling:** Die Datenerhebung ist dynamisch und ändert sich je nach den Erkenntnissen bisher gesammelter Daten.

3. **Ständiges Vergleichen:** Während der Kodierung (siehe weiter unten) entstehen Namen und Definitionen von Konzepten. Diese müssen mit neuen Befunden ständig aktualisiert und alle bisher unter diesem Konzept gestellten Daten erneut darauf überprüft werden, ob sie noch zu dem Konzept passen. Unter Umständen kann es vorkommen, dass ein Konzept geteilt werden muss, weil es zu breit geworden ist.

4. **Unvoreingenommenheit durch andere Theorien:** Dem Anwender der GTM dürfen vorher keine Theorien zu dem untersuchten Thema bekannt sein, weil das dazu führen könnte, dass die Sicht auf die Daten eingeschränkt wird und keine neue Theorie zustande kommt.

Phasen

1. **Offenes Kodieren:** In dieser Phase werden die Daten betrachtet und konzeptualisiert. Dabei wird einem Konzept ein Name gegeben und in einer Beschreibung näher definiert. Diese Beschreibung wird jedes Mal aktualisiert, wenn wieder etwas aus der Datensammlung diesem Konzept zugeordnet wird. Mit der Zeit entstehen mehrere solcher Konzepte.

2. **Axiales Kodieren:** In dieser Phase werden die Konzepte, die in der Phase „Offenes Kodieren“ gefunden wurden, miteinander in Beziehung gebracht, zusammengefasst oder in weitere Konzepte aufgeteilt.

3. **Selektives Kodieren:** In dieser Phase wird ein Konzept bzw. eine Kategorie von Konzepten ausgewählt und in das Zentrum der Betrachtung gestellt. Alle anderen Konzepte werden darum herum positioniert und es wird auf Bedeutungen und Beziehungen zu den anderen Konzepten eingegangen. Ausgehend von dieser Kernkategorie wird dann die „Geschichte“ erzählt.

Nachdem ich mich in die GTM eingearbeitet hatte, habe ich die Dokumentation der Klasse Date der Java-API betrachtet, um Konzepte zu erkennen, die auf Stile bzw. Stileigenschaften von API-Dokumentationen hinweisen. Allerdings hatte ich mich vorher intensiv mit den Wissenstypen von Maalej und Robillard [1] beschäftigt und habe in den Daten nur diese Wissenstypen wiedergesehen. Das vierte Prinzip des GTM, das besagt, dass keine Theorien zu dem Thema bekannt sein dürfen, war in meinem Fall nicht eingehalten. Ich nahm an, dass die Wissenstypen nicht in diese Kategorie fallen, da sie ja „nur“ von verschiedenen Wissenstypen handeln, die in API-Dokumentationen enthalten sein können. Dann habe ich versucht zu betrachten was ein Textabschnitt versucht zu erreichen. Also nicht welche Wissenstypen er dem Leser zu vermitteln versucht, sondern auf einer höheren Ebene, wie diese Wissenstypen vermittelt werden und die API-Dokumentation erklärt wird. Das war allerdings viel anspruchsvoller und ich war mit der Materie noch nicht genug vertraut und deswegen fuhr ich mit der GTM nicht fort. Es gab mir allerdings neue Impulse, wie ich weiterarbeiten sollte und hat mich für meine weiteren Vorgehensweisen inspiriert.

4.3 Gruppendiskussion mit Python-Entwicklern

Während meines Studiums musste ich überwiegend mit Java programmieren und daher habe ich die Arbeits- und Denkweisen, die vielen Java-Entwickler gängig sind, kennengelernt. Ich habe zwar ebenfalls erste Erfahrungen mit Python sammeln können, würde mich aber nicht als fortgeschrittenen Python-Programmierer sehen. Daher war es für mich schwer mir vorzustellen, wie ein Python-Programmierer die Dokumentation der Python-API liest. Dabei ist das ziemlich wichtig, denn so kann ich unter anderem auch besser beurteilen, wie gut ein Element beschrieben ist und, wie die Dokumentation benutzt wird.
Um die Sicht der Python-Programmierer auf API-Dokumentationen und insbesondere auf die Dokumentation der Python-API zu verstehen, habe ich eine Gruppendiskussion durchgeführt. Dazu habe ich am monatlichen Treffen der „Python Users Berlin“[21] (PUB) teilgenommen. Die PUB ist eine Gruppe Berliner Python-Programmierer, die sich über das soziale Netzwerk www.meetup.com, in der sich Interessensgruppen bilden und Meetings veranstalten können, organisieren. Nach einer kurzen Vorstellung und Einführung in das Thema habe ich folgende Fragen an die Wand projiziert und die Diskussion moderiert:

- Warum ist die Dokumentation der Python-API gut/schlecht?
- In welchen Fällen ist sie besonders/kaum hilfreich?
- Was findet ihr an anderen API-Dokumentationen gut/schlecht?

Die Fragen sind alle bewusst offen gestellt und fragen Meinungen und Erfahrungen ab. Dadurch sollten die Teilnehmer der Diskussion nicht eingeengt werden und die Möglichkeit haben sich frei zu äußern bzw. auf die Kommentare der anderen Teilnehmer reagieren zu können. Es waren ca. 70 Leute anwesend, von denen sich ungefähr 15 Leute zu Wort gemeldet haben. Der Konsens war, dass die Dokumentation der Python-API gut sei, was unter anderem daran liegen soll, dass sie über viele Jahre von einer offenen Community gepflegt und verfeinert wurde. Des Weiteren fand die Meinung große Zustimmung, dass Code-Beispiele innerhalb der Dokumentation der Python-API oder generell in API-Dokumentationen sehr hilfreich beim Verstehen der API sind. Viele wünschten sich eine Möglichkeit die Code-Beispiele in der im Internet-Browser geöffneten API-Dokumentation direkt editieren und ausführen zu können.
Einige fanden den Einsatz von Hyperlinks sehr hilfreich, da sie so schneller zu relevanten Teilen der Dokumentation geführt werden. Andere hingegen

[21] https://www.meetup.com/Python-Users-Berlin-PUB/
(letzter Zugriff: 10.05.2017)

wünschten sich bessere Navigationsmöglichkeiten, da die Hyperlinks sie an eine andere Stelle springen lassen und sie somit die Orientierung verlieren. Große Sprünge im Abstraktionsniveau können Leser ebenfalls verwirren, da sie das Gefühl bekommen können unter- bzw. überfordert zu sein oder meinen an eine Stelle gelangt zu sein, die nicht für Nutzer der API gedacht sind, weil z.B. sehr stark auf die Implementierung der API eingegangen wird.
Die 20-minütige Diskussionsrunde hat ihren Zweck erfüllt und mir einen Einblick in die Welt der Python-Entwickler ermöglicht. Der Input aus der Diskussionsrunde war hilfreich dabei bestimmte Beobachtungen über die Dokumentation der Python-API besser deuten zu können, und hat ebenfalls die Theoriebildung unterstützt.

4.4 E-Mailverkehr der Autoren

Mich hat die Frage interessiert, wie die Entwürfe der Java- und Python-API-Dokumentation zustande gekommen sind, welche Veränderungen es im Laufe der Zeit gab und welche Faktoren bei den Entscheidungen eine Rolle gespielt haben.
Da Python schon sehr früh angefangen hat open-source zu sein, sind viele Diskussionen und Designentscheidungsphasen transparent und archiviert. Die Python-Community hat besondere Interessensgruppen (englisch Special Interest Group, kurz SIG) zu unterschiedlichen Bereichen für die Entwicklung und Wartung von Python. Jede dieser SIGs hat einen E-Mail-Verteiler über den kommuniziert wird. Laut dem Archiv des E-Mail-Verteilers von Doc-SIG[22], der Gruppe, die sich um die Erstellung und Wartung der Dokumentation der Python-API kümmert, war die SIG von Februar 1996 bis September 2013 aktiv. In dieser Zeitspanne gab es einen regen E-Mail-Verkehr mit etwa 3900 E-Mails.
Bevor ich meine Fragen an die Mitglieder der Doc-SIG sandte, wollte ich sichergehen, dass die Fragen nicht vor Kurzem schon beantwortet wurden und ich nicht gegen sonstige in der Doc-SIG geltende Regeln und Etikette verstoße. Außerdem war es durchaus möglich, dass die Diskussionen, die meine Fragen beantworten würden, sehr weit zurück in der Vergangenheit geführt wurden und die Leser meiner E-Mail sich nicht mehr daran erinnerten bzw. damals nicht aktiv waren. Aus diesen Gründen habe ich mit der ältesten E-Mail angefangen und mich bis zur neusten E-Mail im Archiv durchgearbeitet und nach für mich relevanten Informationen gesucht. Dabei habe ich anhand des Betreffs entschieden, ob eine E-Mail für mich lesenswert ist oder nicht. Wenn es mehrere Antworten zu einer E-Mail mit einem irrelevanten Betreff gab, überflog ich einige dieser E-Mails, um nachzuschauen, ob die Diskussion sich in eine andere Richtung entwickelte. Nach diesem Prozess verfasste ich eine E-Mail in englischer Fassung, in der ich nach einer kurzen

[22] https://mail.python.org/pipermail/doc-sig/(letzter Zugriff: 10.05.2017)

Einleitung folgende Fragen in englischer Fassung stellte:

1. Wie detailliert sollte die Beschreibung sein, sodass Anfänger nicht im Stich gelassen und Experten nicht gelangweilt werden?

2. Sollte die Nutzung jedes API-Elements(Objekt, Methode, Feld) in einem Code-Beispiel gezeigt werden?

3. Ich erkenne zwei Arten von Beispielen: Kleine Beispiele, die nur die konkrete Nutzung eines API-Elements zeigen, und große Beispiele, in denen mehrere API-Elemente genutzt werden, um etwas umzusetzen. Unter welchen Umständen soll welche Art benutzt werden?

4. Die Dokumentation der Python-API setzt auf viel Text und ähnelt vom Stil her einem Buch. Die Dokumentationen anderer APIs, wie z.B. JavaDoc oder PerlDoc haben eher einen Nachschlagewerk-Charakter. Warum wurde für Python ein anderer Weg gewählt? Wurde das irgendwo diskutiert?

5. Habt ihr sonstige Tipps an Autoren von API-Dokumentationen? Gibt es weitere Dinge, die ich beachten sollte?

Die erste und letzte Frage sind offenere Fragen, da sie Meinungen abfragen. Entsprechend habe ich Antworten erwartet, die mich auf neue Gedanken bringen und mir helfen die Entstehung der Dokumentation der Python-API besser zu verstehen. Die zweite und dritte Frage waren eher konkrete Fragen, deren Antwort vielleicht in einer Anleitung für Autoren steht, die ich nicht kannte und zu der mich jemand hinweisen könnte. Am wichtigsten war jedoch die vierte Frage, da sie direkt Stile anspricht und nach Gründen für die Designentscheidungen der Dokumentation abfragt.
Zu meinem Bedauern musste ich feststellen, dass der Verteiler nicht mehr aktiv war und meine E-Mail nicht an die Empfänger des Verteilers verschickt wurde. Meine Versuche die Verantwortlichen für die Mailingliste zu kontaktieren waren erfolglos. Ich habe danach die Fragen in der Google-Gruppe comp.lang.python[23] gestellt. Es ist ein von Python-Nutzern viel benutztes Forum, um über Python-Themen zu diskutieren. Zwei Nutzer der Gruppe antworteten. Manche Antworten waren vorhersehbar, z.B., dass möglichst alles mit Beispielen abgedeckt werden sollte. Interessant waren die Aussagen, dass der Detailgrad von der Zielgruppe der API abhängt und dass der open-source-Ansatz zu einer heterogenen Dokumentation geführt hat.
Ich hatte eigentlich auf Antworten von den Verantwortlichen für die Dokumentation der Python-API gehofft, aber die konnte ich über die Mailingliste der Doc-SIG nicht erreichen. Dennoch hatte ich einige Denkanstöße von anderen Python-Programmierern erhalten.

[23] https://groups.google.com/forum/#!forum/comp.lang.python
(letzter Zugriff: 10.05.2017)

5 Dokumentationsstile und Stileigenschaften

In diesem Kapitel werden die vorläufigen Ergebnisse meiner Forschung der Java- und Python-API-Dokumentation vorgestellt. Die Hypothese, mit der ich zu Beginn meiner Forschung angetreten bin, lautet, dass es unterschiedliche Stile von API-Dokumentationen gibt und, dass diese Stile über gewisse Eigenschaften verfügen, die je nach Situation unterschiedliche Vor- und Nachteile haben. Eine weitere Annahme war bzw. ist, dass mit einer Liste dieser Eigenschaften und mit den jeweiligen Vor- und Nachteilen Autoren von API-Dokumentationen geholfen werden kann. Ähnlich wie ein Softwareentwickler beim Entwurf seiner Software auf Softwarearchitekturstile oder Entwurfsmuster zurückgreifen kann, um seine Software zu entwerfen, könnte ein Autor einer API-Dokumentation auf die Dokumentationsstile und die Stileigenschaften zurückgreifen und somit eine bessere Dokumentation erstellen.
Um die Dokumentationsstile und die Eigenschaften besser beschreiben zu können, führe ich die Begriffe „Referenzstil“ und „Buchstil“ ein. Dabei hat die Dokumentation der Java-API den Referenzstil und die Dokumentation der Python-API den Buchstil.
Im Folgenden werden Eigenschaften und deren Ausprägung beschrieben, die dazu führen, dass eine API-Dokumentation den Referenzstil oder den Buchstil hat.
Der Aufbau ist immer gleich: Zunächst wird eine Stileigenschaft allgemein beschrieben. Danach wird darauf eingegangen, wie diese Eigenschaft in der Java- bzw. Python-API-Dokumentation ausgeprägt ist. Hierfür werden Beispiele aus den Dokumentationen herangezogen. Für jede Stileigenschaft gibt es außerdem eine kleine Diskussion, in der die Beschränkungen, Vor- und Nachteile, sowie mögliche Folgen für Autoren und Leser besprochen werden.
Dass es sich bei den Stileigenschaften und deren Ausprägungen, die vorgestellt werden, nicht um Einzelfälle handelt, wird mithilfe von Quantifizierungen gezeigt. Dabei wird die Existenz bzw. Häufigkeit der Stileigenschaften und deren Ausprägungen, wenn möglich, immer an den selben fünf Paaren von Dokumentationsseiten gezeigt. Ein Paar besteht jeweils aus einer Dokumentationsseite der Java- bzw. Python-API-Dokumentation. Die Paare wurden so gewählt, dass die Funktionalität innerhalb eines Paars sich ähnelt. Mit dieser Art der Quantifizierung wird eine hohe Glaubwürdigkeit der Ergebnisse erreicht, obwohl relativ wenige Dokumentationsseiten betrachtet werden.
Am Ende des Kapitels werden dann die beiden Dokumentationsstile „Referenzstil“ und „Buchstil“ nochmal gegenübergestellt.

5.1 Aufbau und Form der Dokumentation

Beschreibung Die offensichtlichste und zugleich einflussreichste Eigenschaft einer API-Dokumentation, die den Stil prägt, ist der Aufbau und die Form der API-Dokumentation. Der Aufbau und die Form der API-Dokumentation hat starke Auswirkungen darauf, wie schnell und einfach der Leser sich einen Überblick verschaffen, sich orientieren und somit das gesuchte API-Element finden kann. Außerdem wird dadurch das Navigieren innerhalb der API-Dokumentation beeinflusst.

Java-API Die Struktur und Hierarchie der Java-API-Dokumentation entspricht dem der API. Die Java-API hat in der obersten Hierarchieebene Pakete. Pakete sind Sammlungen von thematisch oder kategorisch verwandten API-Typen(Klassen, Interfaces, ...), die Zugriffsschutz, einfachere Verwaltung von Namensräumen und einfachere Strukturierung ermöglichen. Pakete können verschachtelt sein, sodass sie anderen Paketen untergeordnet sind und selbst Unterpakete haben. Pakete können neben anderen Paketen auch API-Typen (Klassen, Interfaces, ...) enthalten.
API-Typen beinhalten den Quellcode (bzw. Bytecode), der zur Laufzeit ausgeführt wird.
Klassen zum Beispiel sind Blaupausen für Objekte, in denen der Zustand und das Verhalten von Objekten definiert werden. Ähnlich wie Pakete können auch Klassen verschachtelt sein, sodass sie Unter- und Oberklassen haben können. Jede Klasse der Java-API befindet sich in einem Paket. Allgemein können Klassen Konstruktoren für das Erzeugen von Objekten der Klasse, Felder, die den Zustand des Objekts bestimmen und Methoden, die das Verhalten des Objekts bestimmen, enthalten.
Die Struktur der Java-API-Dokumentation orientiert sich an dieser Struktur, mit der Ausnahme, dass Paket- und API-Typ-Hierarchien geebnet dargestellt werden. Die Dokumentation besteht aus drei Ebenen: Die *Übersichtsebene*, die *Paketebene*, die *API-Typ-Ebene*.
In der *Übersichtsebene*(siehe Abbildung 8) sind alle Pakete alphabetisch sortiert. Zu jedem Paketnamen, dem auch die Paket-Hierarchie entnommen werden kann, gehört eine zusammenfassende Beschreibung des Pakets. Die Paketnamen sind gleichzeitig Hyperlinks, die beim Betätigen zur *Paketebene* des jeweiligen Pakets führen.
In der *Paketebene* (siehe Abbildung 9a) sind die API-Typen in diesem Paket aufgelistet. Dabei ist für jede Art von API-Typ eine eigene Liste vorhanden, in der die API-Typen alphabetisch sortiert sind. In der Paketebene ist im Idealfall eine ausführliche Beschreibung, in der das Paket, für das Paket wichtige Konzepte und die Beziehungen zwischen den API-Typen des Pakets beschrieben werden. Bei der Java-API-Dokumentation ist das aber oft nicht der Fall. Stattdessen gibt es eine kurze und grobe Beschreibung des Pakets. Auch hier geben die Namen der API-Typen Auskunft über die Vererbung

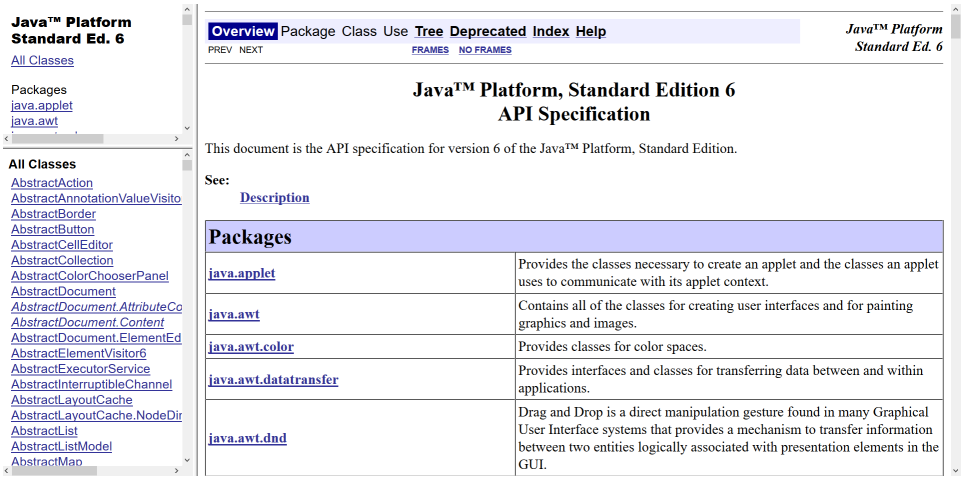
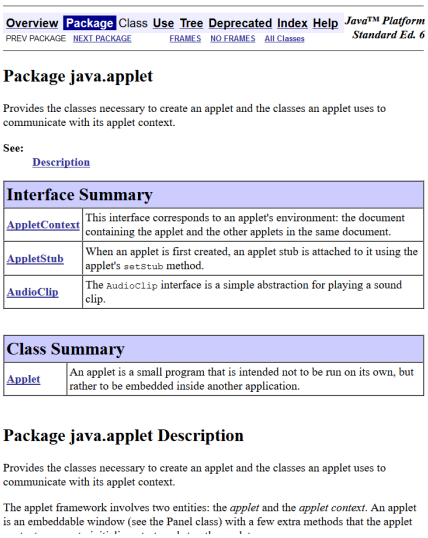


Abbildung 8: Übersichtsebene der Java-API-Dokumentation [Quelle: https://docs.oracle.com/javase/6/docs/api/(letzter Zugriff: 10.05.2017)]

und zu jedem API-Typ gibt es eine zusammenfassende Beschreibung. Die Hyperlinks der Paketnamen führen zur jeweiligen *API-Typ-Ebene*.
In der untersten Ebene - der *API-Typ-Ebene* - (siehe Abbildung 9b) sind Informationen bezüglich der Hierarchie und Vererbung des API-Typs, sowie Informationen über die Elemente des API-Typs vorhanden. Eine Beschreibung des API-Typs gibt Informationen über Sinn und Zweck des API-Typs, erklärt wichtige Konzepte usw. In Listen zu jeder Art von API-Elementen (Feld, Konstruktor, Methode, ...) wird eine Übersicht über die Elemente des API-Typs gegeben. Neben der Signatur der Elemente gibt es eine zusammenfassende Beschreibung des Elements. In den Übersichtslisten sind die Elemente alphabetisch sortiert. Die Betätigung der Hyperlinks führt zu der Beschreibung des Elements. Unter diesen Übersichtslisten sind Listen, die auf geerbte Elemente hinweisen. Die Beschreibungen der Elemente des API-Typs befinden sich auf der gleichen Seite, unter den Übersichtslisten, gruppiert nach Arten von API-Elementen. Zu jedem API-Element gibt es einen Dokumentationsblock. Das ist ein abgegrenzter Bereich, in dem das API-Element beschrieben wird. Die Dokumentationen der Elemente bestehen mindestens aus einer Signatur und einer verbalen Beschreibung. Je nach Element können zusammenfassende Beschreibungen zu Eingabeparametern, Rückgabewerten, möglichen Exceptions und verwandten API-Typen hinzukommen. Außerhalb von diesen Dokumentationsblöcken und der Beschreibung des API-Typs ganz oben, ist für ein API-Element keine Beschreibung vorgesehen.



Overview Package Class Use Tree Deprecated Index Help *Java™ Platform Standard Ed. 6*
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES All Classes

Package java.applet

Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.

See:
Description

Interface Summary	
AppletContext	This interface corresponds to an applet's environment: the document containing the applet and the other applets in the same document.
AppletStub	When an applet is first created, an applet stub is attached to it using the applet's `setStub` method.
AudioClip	The `AudioClip` interface is a simple abstraction for playing a sound clip.

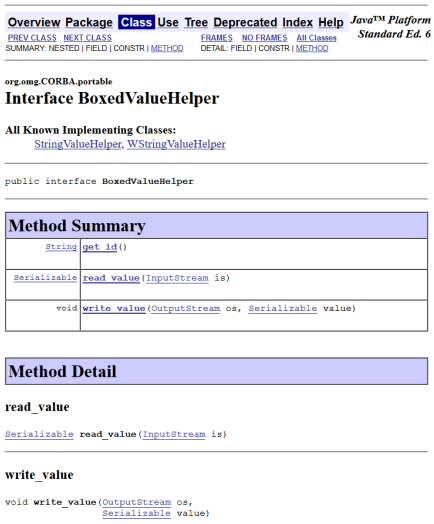
Class Summary	
Applet	An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.

Package java.applet Description

Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.

The applet framework involves two entities: the *applet* and the *applet context*. An applet is an embeddable window (see the Panel class) with a few extra methods that the applet

(a) Paketebene des Pakets java.applet [Quelle: http://docs.oracle.com/javase/6/docs/api/java/applet/package-summary.html (letzter Zugriff: 10.05.2017)]



Overview Package Class Use Tree Deprecated Index Help *Java™ Platform Standard Ed. 6*
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.omg.CORBA.portable
Interface BoxedValueHelper

All Known Implementing Classes:
StringValueHelper, WStringValueHelper

```
public interface BoxedValueHelper
```

Method Summary	
String	get_id()
Serializable	read_value(InputStream is)
void	write_value(OutputStream os, Serializable value)

Method Detail

read_value

```
Serializable read_value(InputStream is)
```

write_value

```
void write_value(OutputStream os,
                 Serializable value)
```

(b) API-Typ-Ebene des Interfaces BoxedValueHelper [Quelle: https://docs.oracle.com/javase/6/docs/api/org/omg/CORBA/portable/BoxedValueHelper.html (letzter Zugriff: 10.05.2017)]

Abbildung 9: Paket- und API-Typ-Ebene

Python-API Verglichen mit der Java-API-Dokumentation hat die Struktur und Hierarchie der Python-API-Dokumentation weniger Gemeinsamkeiten mit der Struktur der API.
Anders als in der Java-API, gibt es in der obersten Hierarchieebene der Python-API neben Paketen auch API-Typen. Pakete in Python sind sehr ähnlich zu Paketen in Java. Anders sieht es bei den API-Typen aus. Während es bei Java mehrere API-Typen gibt, gibt es bei Python nur den API-Typ „Modul“. Innerhalb eines Moduls sind API-Elemente von Python (Klassen, Funktionen, Felder, ...) implementiert.
Die Dokumentation besteht ebenfalls aus drei Ebenen, die sich aber von den drei Ebenen der Java-API-Dokumentation unterscheiden: Die *Übersichtsebene*, die *Kapitelebene*, die *API-Typ-Ebene*.
Die *Übersichtsebene* der Python-API-Dokumentation liefert zuerst eine einleitende Beschreibung der API und der API-Dokumentation, die in einem Einführungskapitel weiter vertieft wird. Anschließend folgt wie in einem Buch ein Inhaltsverzeichnis. Denn anders als bei der Java-API-Dokumentation, ist die Python-API-Dokumentation wie ein Buch in Kapitel unterteilt. Jedes Kapitel hat ein anderes Thema und die Reihenfolge der Kapitel ist nicht alphabetisch, sondern thematisch. Die ersten Kapitel sind die wichtigsten und grundlegend für das Verständnis der API. Je weiter unten ein

Kapitel ist, desto spezieller und weniger wichtig ist es, um ein Gesamtverständnis über die API zu erhalten. Bereits in der Übersichtsseite sind die Unterkapitel, samt kurzen Beschreibungen zu sehen (siehe Abbildung 10). Die Hyperlinks der Kapitel führen zu der Beschreibung des Kapitels und somit zu der *Kapitelebene*, wohingegen die Hyperlinks der Unterkapitel direkt zur Beschreibung eines Moduls oder direkt zum Unterkapitel eines Kapitels führen.

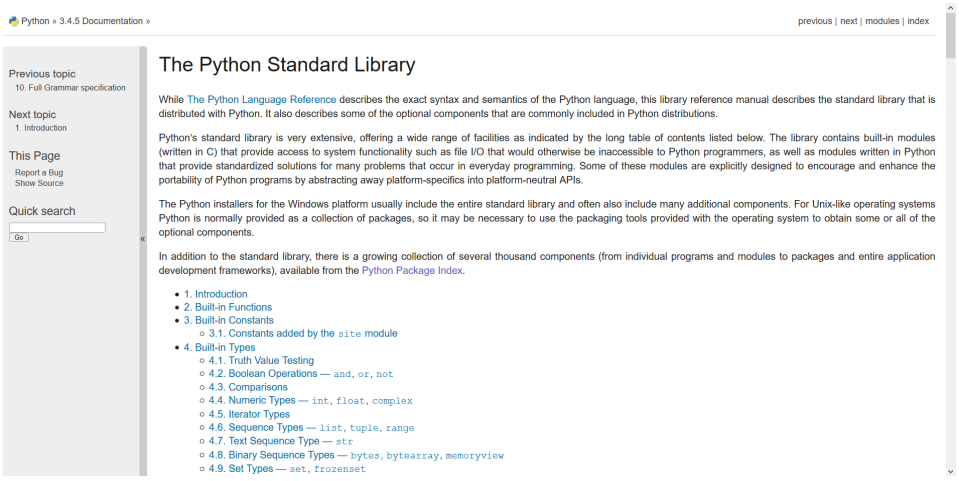
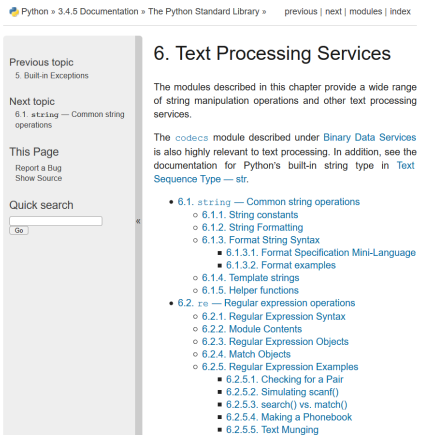
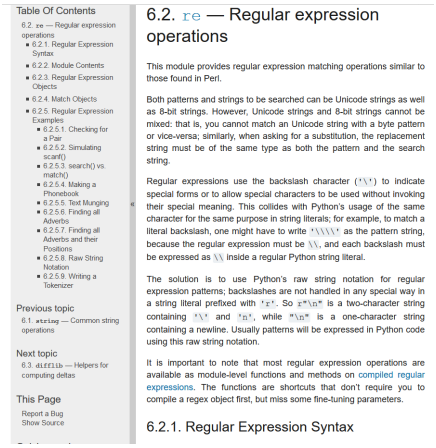


Abbildung 10: Übersichtsebene der Python-API-Dokumentation [Quelle: https://docs.python.org/3.4/library/index.html(letzter Zugriff: 10.05.2017)]

Die *Kapitelebene* (siehe Abbildung 11a) enthält eine Beschreibung darüber, worüber das Kapitel handelt. Außerdem zeigt es wieder ein Inhaltsverzeichnis, in dem die Unterkapitel enthalten sind, die meistens Module sind und Unterunterkapitel, die direkt zu Stellen der untersten Ebene - der *API-Typ-Ebene*- führen.
Die *API-Typ-Ebene* (siehe Abbildung 11b) bildet die eigentliche Dokumentationsseite, in der ein Modul (idealerweise) mit all seinen Elementen beschrieben wird. Anders als in der Java-API-Dokumentation können die Erklärungen API-Elemente in einer beliebigen Reihenfolge und über die gesamte Seite verteilt sein. Es gibt keine fest vorgeschriebenen Bereiche, in denen bestimmte Inhalte beschrieben werden müssen. Meist gibt es zu Beginn eine ausführliche Erklärung darüber, was das Modul macht, welche Bestandteile es hat und wie die Beziehungen der Elemente zueinander sind. Anschließend gibt es Abschnitte, in denen thematisch zusammengehörende Elemente beschrieben werden. Ein Inhaltsverzeichnis auf der linken Seite soll für eine bessere Übersicht sorgen.



(a) Kapitelebene des Kapitels 6. Text Processing Services[Quelle: https://docs.python.org/3.4/library/text.html (letzter Zugriff: 10.05.2017)]



(b) API-Typ-Ebene des Moduls `re` [Quelle: https://docs.python.org/3.4/library/re.html (letzter Zugriff: 10.05.2017)]

Abbildung 11: Kapitel- und API-Typ-Ebene

Diskussion Dadurch, dass die Struktur der Java-API-Dokumentation dem der Java-API ähnelt, kann ein Leser, der sich mit der Struktur der API auskennt, viel schneller die gesuchte Dokumentationsseite erreichen, als bei der Python-API-Dokumentation. Voraussetzung dafür ist aber, dass er weiß, in welcher Dokumentationsseite bzw. in welchem API-Typ er nachschauen muss. Wenn der Leser diese Informationen nicht hat, wird es ihm schwer fallen die richtige Stelle zu entdecken. Gute Paketnamen, Paketbeschreibungen in der Übersichtsebene und gute Paketbeschreibungen, in denen ein Überblick über die API-Typen eines Pakets gegeben wird, können hier Abhilfe schaffen.
Die Struktur der Python-API-Dokumentation hat eine didaktische Funktion. Sie signalisiert dem Leser, welche Bestandteile der Dokumentation wichtiger sind und welche Elemente der Dokumentation dafür gedacht sind oft zusammen genutzt zu werden bzw. gemeinsam für eine bestimmte Sache verwendet werden können. Dadurch, dass die Dokumentationsseiten in keine starren Formen gepresst werden müssen, können sie von den Autoren so aufgebaut werden, wie sie es für am besten halten, um die API-Elemente bestmöglich erklären zu können. Das sorgt für eine potenziell bessere bzw. verständlichere Dokumentation für jemanden, der sich mit der API wenig auskennt und sich die Zeit nimmt, um viel zu lesen. Denn dadurch, dass die Form wenig Auskunft über die Struktur gibt, muss mehr im Text darauf eingegangen werden. Dadurch kann aber der Leser besser informiert und angeleitet werden.

Der Aufbau und die Form der Java-API-Dokumentation ist für große APIs mit vielen API-Typen und -Elementen besser geeignet. Denn durch die alphabetische und hierarchische Form, die in der Übersichtsseite nur die oberste Hierarchie anzeigt, wirkt die Dokumentation viel übersichtlicher. Außerdem ist sie besser für Nutzer, die sich mit der API gut auskennen, aber schnell was nachschauen möchten.
Der Aufbau und die Form der Python-API-Dokumentation ist besser für kleinere APIs mit weniger API-Typen und -Elementen. Denn es wird mehr und detaillierter auf diese eingegangen. Aufbau und Form der Python-API-Dokumentation setzen eher darauf, dass alles bzw. viel gelesen und dabei chronologisch vorgegangen wird. Bei sehr großen APIs bedeutet das sehr viel Text. Dies kann Leser abschrecken und bedeutet mehr Aufwand für die Autoren, um die Dokumentation zu erstellen und zu warten. Dadurch, dass mehr auf Einzelheiten und Zusammenhänge eingegangen wird, ist er eher für Nutzer geeignet, die sich nicht so gut auskennen, oder die ein tiefes Verständnis über die gesamte API haben möchten.

5.2 Beispielnutzung

Beschreibung In vielen der in Kapitel 3 vorgestellten Studien wurde herausgefunden, dass Code-Beispiele sehr hilfreich für das Verstehen einer API sind. Unabhängig davon, ob es sich um Leser handelt, die sich mit der Programmiersprache oder der API gut auskennen oder nicht, kann das Verhalten eines API-Elements durch ein Code-Beispiel einfach erläutert werden. Dabei können Code-Beispiele auf unterschiedliche Weisen eingesetzt werden. Kurze Code-Beispiele können dabei helfen das Verhalten eines einzelnen API-Elements zu beschreiben. Längere Code-Beispiele können bestimmte Anwendungsfälle implementieren. Solche vollständigen Code-Blöcke können von Lesern der Dokumentation direkt in ihren Quellcode hineinkopiert und mit einigen Anpassungen in ihre Software eingebunden werden. Dieses Vorgehen wird auch „debugging into existence“ genannt.

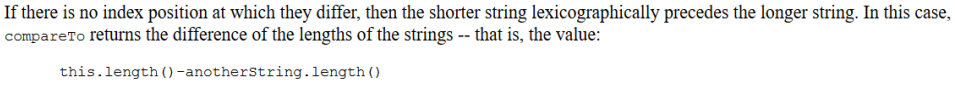
Java-API Die Entwickler der Java-API sehen die Dokumentation der Java-API eher als eine Spezifikation, als eine Anleitung. Das geht aus einer offiziellen Anleitung für die Erstellung von Dokumentation für die Java-API[24] hervor. In dieser Anleitung steht, wie mithilfe des JavaDoc-Werkzeugs Dokumentation für die Java-API erstellt werden kann. Ferner steht in der Anleitung:

> „*We spend time and effort focused on specifying boundary conditions, argument ranges and corner cases rather than defining*

[24] http://www.oracle.com/technetwork/java/javase/documentation/writingdoccomments-137785.html (letzter Zugriff: 10.05.2017)

> *common programming terms, writing conceptual overviews, and including examples for developers.*“

In einer anderen Anleitung, in der mehr auf den semantischen Inhalt von Java-API-Dokumentation eingegangen wird[25], wird aber die Nutzung von Beispiel-Code als nützlich und illustrativ beschrieben und empfohlen.
In der Java-API-Dokumentation sind wenige Code-Beispiele vorhanden, wie eine Studie von Maalej und Robillard gezeigt hat (siehe Abbildung 4 in Kapitel 3.3). Dabei handelt es sich um kurze Code-Schnipsel, die z.B. den Aufruf eines API-Elements illustrieren und in die verbale Beschreibung eines API-Elements eingebettet sind. Abbildung 12 zeigt, wie mittels Java-Code der zuvor in Worten erklärte Teil nochmal illustriert wird.


If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

```
this.length()-anotherString.length()
```

Abbildung 12: Beispielschnipsel in der Beschreibung der compareTo-Methode der String-Klasse in der Java-API [Quelle: https://docs.oracle.com/javase/6/docs/api/java/lang/String.html(letzter Zugriff: 10.05.2017)]

Python-API Dass es in der Python-API-Dokumentation mehr Beispiele gibt, als in der Java-API-Dokumentation, hat Wildermann in seiner Bachelorarbeit gezeigt (siehe Abbildung 5 in Kapitel 3.3). Da Python open-source-basiert ist und intensiv auf Beiträge Freiwilliger setzt, hat es eine sehr ausführliche Anleitung für Personen, die an Pythons Entwicklung teilhaben möchten. In der „Python Developer's Guide“[26] wird ausführlich erklärt, wie jemand bei der Entwicklung von Python helfen kann. Es gibt auch ein ausführliches Kapitel[27], in dem erklärt wird, wie die Python-API zu dokumentieren ist. Das Kapitel beinhaltet sowohl technische Informationen darüber mit welchen Werkzeugen und Methoden das Erstellen der Dokumentation funktioniert, als auch Informationen darüber wie der Inhalt gestaltet werden soll. Es wird propagiert, dass Code-Beispiele oft verständlicher sind als verbale Beschreibungen. Die Bedeutung von sinnvollen Anwendungsfällen, die mittels Code-Beispiele umgesetzt werden sollen, wird unterstrichen. Es gibt jedoch keine Aussagen darüber, wo Beispiele platziert werden sollen und wie intensiv Code-Beispiele genutzt werden sollen. In den E-Mail-Archiven der

[25] http://www.oracle.com/technetwork/java/javase/documentation/index-142372.html(letzter Zugriff: 10.05.2017)
[26] https://docs.python.org/devguide/index.html
(letzter Zugriff: 10.05.2017)
[27] https://docs.python.org/devguide/documenting.html
(letzter Zugriff: 10.05.2017)

Autorengruppe der Python-API-Dokumentation (siehe Kapitel 4.4) bin ich auf ein Latex-Template[28] gestoßen. Es wurde von Guido van Rossum, dem Erfinder von Python, geschrieben und beinhaltete Kommentare für das Erstellen von Dokumentationen. In diesem Template ist eine grobe Gliederung für eine Dokumentationsseite enthalten und gegen Ende wird empfohlen ein komplettes Beispiel anzugeben. Dieses Template stammt aus der Mitte der 90er Jahre. In dieser Zeit wurden Dokumentationen mit Latex erstellt. Heutzutage wird die Auszeichnungssprache reStructuredText verwendet [29]. In der Python-API-Dokumentation gibt es immer wieder „Examples"-Abschnitte, in denen größere Blöcke von Code-Beispielen enthalten sind (siehe Abbildung 13).

18.1.5. Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print('Connected by', addr)
while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall(b'Hello, world')
data = s.recv(1024)
s.close()
print('Received', repr(data))
```

Abbildung 13: Beispiel-Blöcke in der Dokumentation des socket-Moduls in der Java-API [Quelle: https://docs.python.org/3.4/library/socket.html(letzter Zugriff: 10.05.2017)]

Dabei ist es üblich in solchen großen Beispielblöcken gängige Anwendungsfälle oder die Nutzung der API-Elemente und deren Unterschiede zu zeigen.

Diskussion Beispiele werden in vielen Lebensbereichen dazu verwendet, um einen Sachverhalt verständlicher zu machen. Gerade bei abstrakten Themen kann ein Beispiel maßgeblich dabei helfen etwas zu verstehen. Im Mathematikunterricht, unabhängig davon, ob in der Grund-, Ober- oder Hochschule, werden deshalb Beispiele verwendet, um die Gesetze der Mathematik

[28] https://svn.python.org/projects/python/tags/proof1/Doc/libtemplate.tex(letzter Zugriff: 10.05.2017)

[29] http://docutils.sourceforge.net/rst.html(letzter Zugriff: 10.05.2017)

zu verdeutlichen. Mehrere Studien haben gezeigt, dass Code-Beispiele von API-Nutzern gerne dazu genutzt werden, um die API zu verstehen.
Kleine Code-Beispielschnipsel haben den Vorteil, dass sie in die verbale Beschreibung bzw. zum Dokumentationsblock eines API-Elements eingebettet werden können, ohne die Form bzw. die Struktur zu stören. Sie können dabei helfen das Element verständlich zu machen. Es kann aber auch sein, dass sie den Leser verwirren, weil eigentlich vorher noch ganz viel Code geschrieben werden müsste, damit dieser Code-Beispielschnipsel funktioniert.
Kleine Code-Beispielschnipsel, die so eingebettet sind, können auch dafür verwendet werden, um auf ähnliche Methode zu verweisen bzw. die Funktionalität eines API-Elements mithilfe eines anderen API-Elements zu erklären. Die Methode `gibLängeInMeter(Objekt)`, die zu einem gegebenen `Objekt` die Länge in Meter ausgibt, kann z.B. mit der Beschreibung:

> Macht das gleiche wie: `gibLängeInZentimeter(Objekt)/100`

beschrieben werden, wobei die Methode `gibLängeInZentimeter(Objekt)` die Länge des Eingabeparameters `Objekt` in Zentimeter zurückgibt.
Große Code-Beispielblöcke haben den Vorteil, dass sie Anwendungsfälle sehr ausführlich implementieren können. Das sorgt nicht nur dafür, dass sie ein tieferes Verständnis über die Beziehungen bzw. das Zusammenspiel der Elemente eines API-Typs geben können, sondern auch, dass dem Leser auf schnelle Art und Weise geholfen werden kann sein Problem zu lösen. Dieser kann nämlich dieses Code-Beispiel übernehmen und es seinen Anforderungen entsprechend anpassen.
Große Code-Beispielblöcke können auch dafür verwendet werden, um hintereinander Methoden mit unterschiedlichen Ein- und Ausgaben oder die Ausgaben von mehreren ähnlichen Methoden zu zeigen, damit die Unterschiede dieser Methoden verdeutlicht werden.
Ein Nachteil großer Code-Beispielblöcke kann sich ergeben, wenn diese sehr oft in einer API-Dokumentation verwendet werden. Ein Nutzer kann auf der Suche nach einer für ihn relevanten Dokumentationsseite dazu verleitet werden sich nur die Code-Beispielblöcke anzugucken, da er fälschlicherweise den Eindruck bekommen hat, dass in diesen Code-Beispielblöcken alle Elemente, die in einer Dokumentationsseite vorhanden sind, gezeigt werden. Es kann aber durchaus sein, dass ein API-Element, z.B. eine Methode, in einer Dokumentationsseite beschrieben wird, aber nicht in den Code-Beispielblöcken vorkommt. Der Leser, der sich nur auf die Code-Beispielblöcke fokussiert, kann so seine Suche unnötig verlängern.
Ein weiterer Nachteil ist, dass die Beispiele nach einer Aktualisierung der API nicht mehr funktionieren können, weil sich das Verhalten bestimmter API-Elemente verändert hat oder weil bestimmte API-Elemente nicht mehr existieren. Das erhöht den Aufwand, die Dokumentation aktuell zu halten.

In der Python-API-Dokumentation werden sowohl kleine Code-Beispielschnipsel als auch große Code-Beispielblöcke verwendet, um ein tieferes und Verständnis der API zu gewährleisten sowie den Lesern ein schnelleres Lösen ihrer Probleme zu ermöglichen.
In der Java-API-Dokumentation wird eher auf beide Arten von Code-Beispielen verzichtet, da diese der Dokumentation einen „Anleitungs"- bzw. „Tutorial"-Charakter verleihen und die uniforme, starre Struktur beschädigen. Letzteres passiert besonders bei großen Code-Beispielblöcken, weshalb auf diese eher verzichtet wird. Im Falle von Java wären große Code-Beispielblöcke noch größer, weil in mehreren Zeilen davor Objekte erzeugt und eingestellt werden müssen. Bei Python geht das kürzer.

Quantifizierung Tabelle 4 zeigt die Anzahl der Code-Beispielblöcke und -Beispielschnipsel der fünf verglichenen Paare. Die mittlere Spalte zeigt die Anzahl der Code-Beispielblöcke. In den untersuchten Dokumentationsseiten der Java-Klassen sind keine, wohingegen in jeder untersuchten Dokumentationsseite der Python-Module Code-Beispielblöcke vorhanden sind. Die *-Markierung bedeutet, dass es an mehreren Stellen der Dokumentationsseite Code-Beispielblöcke gibt. Die **-Markierung bedeutet, dass es extra einen Abschnitt gibt, in dem mehrere Code-Beispielblöcke aufgeführt werden. Diese beiden Markierungen gibt es nur bei den betrachteten Python-API-Dokumentation.
Die rechte Spalte zeigt die Anzahl der Code-Beispielschnipsel. Auch hier wird in der Python-API-Dokumentation mehr Gebrauch davon gemacht.
Die ***-Markierung bedeutet, dass auch Pseudo-Code dazu benutzt wurde, um die Funktionalität eines API-Elements zu beschreiben.
Die Quantifizierung deutet darauf hin, dass in der Python-API-Dokumentation Beispielblöcke und Beispielschnipsel eine viel stärkere Verwendung haben, als in der Java-API-Dokumentation.

5.3 Lokalität

Im Gegensatz zu systematischen Programmierern, sind pragmatische und opportunistische Programmierer nicht daran interessiert ganze Dokumentationsseiten zu lesen, sondern nur ihr aktuelles Problem zu lösen, um wieder schnell mit ihrer Programmiertätigkeit fortzufahren [17]. Das heißt sie suchen die Beschreibung des API-Elements, mit dem sie gerade ein Problem haben oder das sie brauchen, um etwas in ihrem Quellcode umzusetzen. Für solch ein Vorgehen ist es wichtig, dass alle kritischen Informationen eines API-Elements kompakt bzw. direkt in der Beschreibung des API-Elements vorhanden sind. Kritische Informationen sind Informationen, die für eine erfolgreiche Verwendung eines API-Elements unabdingbar sind.
Muss außerhalb der Beschreibung eines API-Elements gelesen werden, um das API-Element verwenden zu können?

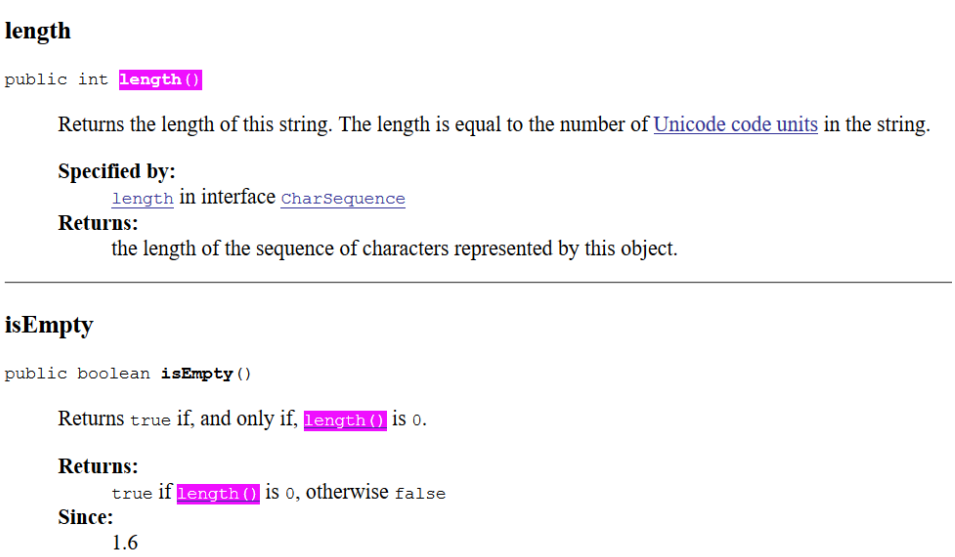
API-Typ	**Code-Beispielblock**	**Code-Beispielschnipsel**
Calendar	0	Viele in der Klassenbeschreibung Viele in Elementbeschreibungen
datetime	12*,**,***	Sehr viele
Socket	0	Keine in der Klassenbeschreibung Wenige in Elementbeschreibungen
socket	9*,**	Sehr viele
Random	0	Keine in der Klassenbeschreibung Viele in Elementbeschreibungen***
random	3**	2-3 Stück
ZipFile	0	0
zipfile	2	Viele
String	0	Wenige in der Klassenbeschreibung Viele in Elementbeschreibungen***
string	9*,**	Viele

Tabelle 4: Anzahl der Code-Beispielblöcke und Code-Beispielschnitzel. Die oberen Paarkomponenten sind API-Typen der Java-API und die unteren der Python-API. *: Code-Beispielblöcke über die Dokumentationsseite verteilt. ** Abschnitt für Code-Beispielblöcke. ***: Pseudo-Code

Stehen außerhalb der Beschreibung eines Elements Informationen, die in der Beschreibung des API-Elements fehlen, obwohl sie darin enthalten sein müssten?
Das sind die Fragen, die die Lokalität im Wesentlichen ausmachen. Je weniger außerhalb der Beschreibung eines Elements gelesen werden muss, um das API-Element verwenden zu können bzw. je weniger Informationen außerhalb der Beschreibung eines API-Elements stehen, die in der Beschreibung des API-Elements hätten stehen müssen, desto höher ist die Lokalität eines API-Elements.

Java-API Dadurch, dass jedes API-Element in einer Dokumentationsseite der Java-API-Dokumentation seinen eigens für ihn vorgesehenen, abgegrenzten Bereich hat, gibt es außerhalb dieses Bereichs kaum wichtige Informationen über das API-Element. Bei besonders komplexen API-Typen kann es sein, dass in der API-Typ-Beschreibung (z.B. Klassenbeschreibung) Informationen darüber existieren, welche Schritte getan werden müssen, bevor ein API-Element eingesetzt werden kann. Das zeichnet sich dadurch aus, dass der Name eines API-Elements meist nur in der Übersichtstabelle, dem Dokumentationsblock des API-Elements und in „See also“-Abschnitten in Dokumentationsblöcken anderer API-Elemente auftaucht. Taucht der Name eines API-Elements A in der API-Typ-Beschreibung (z.B. Klassenbeschrei-

bung) oder in dem Dokumentationsblock eines API-Elements B auf, ist es für das Verständnis des API-Elements A nicht von Bedeutung, sondern dient dazu das API-Element B besser erklären zu können(siehe Abbildung 14). Und wenn doch, ist diese Information ebenfalls in dem Dokumentationsblock des API-Elements A enthalten.
Beispiel: Die Methode `length()` aus der Klasse `java.lang.String`[30] wird in der Dokumentationsseite 13 Mal erwähnt. Sie taucht zwei Mal in der Methodenübersichtsliste auf, weil sie selber einen Eintrag in dieser Liste hat und weil sie in der zusammengefassten Methodenbeschreibung der Methode `isEmpty()` erwähnt wird. Ein Mal taucht sie in ihrem eigenen Dokumentationsblock in der Signatur auf. Weitere zwei Mal in der Methodenbeschreibung der `isEmpty()`-Methode, in der die `length()`-Methode dazu verwendet wird, um die `isEmpty()`-Methode zu erklären. Aus dem selben Grund wird sie weitere acht Mal in den Dokumentationsblöcken der Methoden `chartAt`(int index), `codePointAt`(int index), `compareTo`(String anotherString), `indexOf`(String str, int fromIndex), `lastIndexOf`(String str), `lastIndexOf`(String str, int fromIndex) und `subSequence`(int beginIndex, int endIndex) erwähnt.



length

`public int length()`

Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Specified by:
`length` in interface `CharSequence`
Returns:
the length of the sequence of characters represented by this object.

---

isEmpty

`public boolean isEmpty()`

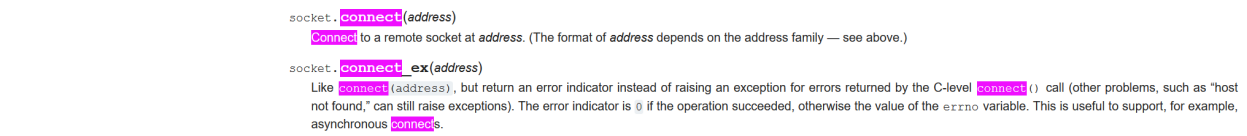
Returns `true` if, and only if, `length()` is `0`.

Returns:
`true` if `length()` is `0`, otherwise `false`
Since:
1.6

Abbildung 14: Die `length()`-Methode wird in dem Dokumentationsblock der `isEmpty()`-Methode erwähnt, um die Funktionalität zu erklären.[Quelle: https://docs.oracle.com/javase/6/docs/api/java/lang/String.html(letzter Zugriff: 10.05.2017)]

[30] https://docs.oracle.com/javase/6/docs/api/java/lang/String.html#length() (letzter Zugriff: 10.05.2017)

Python-API Anders als bei der Java-API-Dokumentation werden API-Elemente nicht immer an nur einem Ort erklärt. Es kommt auch öfter vor, dass die Beschreibung eines API-Elements über mehrere Orte einer Dokumentationsseite verteilt ist. Außerdem gibt es oft keinen abgegrenzten Bereich für die Beschreibung von Klassen bzw. Objekten. Das ermöglicht eine Vermischung der Beschreibungen von Objekten und ihren Methoden bzw. Attributen. Die Klasse `date` im `datetime`-Modul[31] der Python-API ist ein Beispiel dafür. Nachdem in einer einleitenden Beschreibung erklärt wird, was die Klasse tut, wird der Klassenkonstruktor mit seinen Parametern vorgestellt. Dann folgen Methoden, die ebenfalls *date*-Objekte erstellen. Dann folgen Attribute. Bis hierher ist es ähnlich wie in der Java-API-Dokumentation aber dann kommen wieder Informationen über `date`-Objekte, gefolgt von Methodenbeschreibungen und Beispielblöcken. Würde es sich um eine Java-API-Dokumentation handeln, würden die Informationen über die `date`-Objekte nicht einmal am Anfang und einmal zwischen den Methoden- und Attributbeschreibungen platziert, sondern nur am Anfang.
Obwohl Methodenbeschreibungen oft kompakter gehalten werden und in der Regel nicht so verteilt erklärt werden, kommt es vor, dass es wichtige Informationen zu Methoden gibt, die nur außerhalb des Dokumentationsblocks der Methode stehen. Manchmal wird mit Hyperlinks auf diese Stellen verwiesen, was zwar der Lokalität schadet aber nicht so schlimm ist. Kritischer ist es, wenn wichtige Informationen woanders stehen und der Leser nicht darüber informiert wird.


socket.connect(address)
Connect to a remote socket at address. (The format of address depends on the address family — see above.)

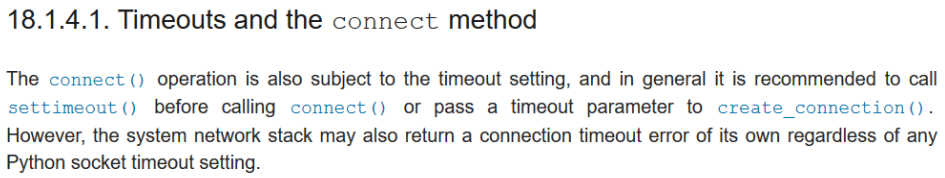
socket.connect_ex(address)
Like connect(address), but return an error indicator instead of raising an exception for errors returned by the C-level connect() call (other problems, such as "host not found," can still raise exceptions). The error indicator is 0 if the operation succeeded, otherwise the value of the errno variable. This is useful to support, for example, asynchronous connects.

Abbildung 15: Dass die Methode `connect`(address) Exceptions wirft, ist statt dem Dokumentationsblock der Methode `connect`(address), dem darunter liegenden Dokumentationsblock der Methode `connect_ex`(address) zu entnehmen [Quelle: https://docs.python.org/3.4/library/socket.html(letzter Zugriff: 10.05.2017)]

Abbildung 15 zeigt so einen Fall. Die Methode `connect`(address) im `socket`-Modul[32] wirft eine Exception, wenn die im Hintergrund aufgerufene C-Funktion `connect()` einen Fehler zurückgibt. Diese Information erfährt der Leser aber nicht, wenn er nur den Dokumentationsblock von `connect`(address) liest, sondern erst, wenn er auch die von `connect_ex`(address) liest. In diesem Fall liegt die wichtige Information gleich im Dokumentationsblock

[31] https://docs.python.org/3.4/library/datetime.html#datetime.date(letzter Zugriff: 10.05.2017)
[32] https://docs.python.org/3.4/library/socket.html#socket.socket.connect(letzter Zugriff: 10.05.2017)

darunter und kann deshalb vom Leser noch aufgeschnappt werden. Es gibt aber eine weitere wichtige Information, die die Methode `connect`(address) betrifft, die ebenfalls nicht im dazugehörigen Dokumentationsblock vorhanden ist. Weiter unten in der Dokumentationsseite gibt es den Abschnitt „18.1.4.1. Timeouts and the connect method“. In diesem Abschnitt werden sehr wichtige Informationen in Verbindung mit `connect`(address) genannt (siehe Abbildung 16). Es gibt zwar Hyperlinks, die von diesem Abschnitt zum Dokumentationsblock der Methode `connect`(address) führen, aber ein Verweis in entgegengesetzter Richtung wäre wichiger. Diese Information ist im Dokumentationsblock der Methode `connect`(address) nicht vorhanden und ein Verweis fehlt ebenfalls.



18.1.4.1. Timeouts and the `connect` method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Abbildung 16: Wichtige Informationen über die Methode `connect`(address), die aber in dem entsprechenden Dokumentationsblock nicht stehen. Es ist auch kein Verweis vorhanden, der den Leser zu dieser Stelle führt.[Quelle: https://docs.python.org/3.4/library/socket.html(letzter Zugriff: 10.05.2017)]

Diskussion Zusammengefasst gibt es vier Formen der Lokalität:

1. Alle wichtigen Informationen über ein API-Element stehen in seinem Dokumentationsblock
2. Alle wichtigen Informationen über ein API-Element stehen entweder in seinem Dokumentationsblock oder an einer anderen Stelle, auf die aber im Dokumentationsblock verwiesen wird.
3. Zu einem API-Element gibt es mehrere Dokumentationsblöcke bzw. unterschiedliche Aspekte eines API-Elements wird an unterschiedlichen Stellen erklärt.
4. Es gibt wichtige Informationen über ein API-Element, die irgendwo außerhalb des Dokumentationsblock stehen und auf die aus dem Dokumentationsblock heraus nicht verwiesen wird.

Die erste Form sorgt für die höchste Lokalität, die dritte und die vierte Form die niedrigste. Die erste Form ist in beiden Dokumentationen enthalten, aber überwiegend in der Dokumentation der Java-API. Die zweite Form ist in beiden API-Dokumentationen selten vorhanden. Die dritte Form ist in der Java-

API-Dokumentation aufgrund der Struktur und Form derselben nicht enthalten, dafür aber in der Python-API-Dokumentation. Die vierte Form bildet eine Ausnahme und kann in beiden Dokumentationen vorkommen. Grob gesagt setzt die Java-API-Dokumentation eher auf eine hohe Lokalität und die Python-API-Dokumentation auf eine etwas niedrigere. Das führt dazu, dass die Python-API-Dokumentation gründlicher gelesen werden muss, um alle wichtigen Informationen zu erhalten. Bei der Java-API-Dokumentation erscheint der Eindruck, dass der Leser die relevanten Informationen über ein API-Element schneller finden kann, weil sie nicht verteilt, sondern an einer Stelle sind. Das kann den Leser aber dazu verleiten, übereifrig zu einer Funktionsbeschreibung zu springen, weil er meint es verstanden zu haben, wie es einzusetzen ist und dann Fehler macht, weil er den Aufbau der API nicht kennt. Das ist bei der Python-API-Dokumentation unwahrscheinlicher. Hier wird der Leser nämlich eher dazu angehalten sich intensiver mit der Dokumentationsseite zu befassen und im Idealfall sogar von Anfang bis Ende alles zu lesen. Die Java-API-Dokumentation ermöglicht also ein schnelleres Auffinden und Verstehen der relevanten Informationen bei einem höheren Risiko die API nicht beim ersten Anlauf beibringen zu können. Die Python-API-Dokumentation versucht dem Leser möglichst die API beizubringen und erschwert das schnelle Auffinden der relevanten Informationen, da an mehreren Stellen gesucht werden muss. Ein fähiger Leser kann aber beim Überfliegen der Dokumentationsseite die relevanten Stellen der API verstehen und die für sein API-Element wichtigen Informationen schnell aufspüren.

Quantifizierung Für die Quantifizierung wurden wieder die fünf Paare untersucht, die schon bei der Quantifizierung des Stils *Beispielnutzung* verwendet wurden. Tabelle 5 zeigt die Ergebnisse. Es wurde für jede Dokumentationsseite überprüft, wie oft welche Form auftritt. Das geschah so:
Konnte zu einem API-Element kein Dokumentationsblock erkannt werden, weil es an mehreren Stellen erklärt wird, hat das API-Element die dritte Form bekommen.
Die Dokumentationsblöcke der API-Elemente wurden der Reihe nach angeguckt. Wenn in der Beschreibung an eine Stelle verwiesen wird, die der Leser lesen muss, um die Funktionalität oder andere wichige Informationen über das API-Element zu verstehen, handelt es sich um Form 2.
Anschließend wurde der Name des API-Elements markiert und alle Vorkommnisse in der Dokumentationsseite hervorgehoben. Dann wurde überprüft, ob das API-Element irgendwo außerhalb des Dokumentationsblock des API-Elements enthalten ist. Wenn ja, wurde überprüft, ob es sich um eine wichtige Information handelt und ob diese wichtige Information auch im Dokumentationsblock des API-Elements ist. Wenn es außerhalb des Dokumentationsblocks der API-Elements eine wichtige Information gibt, die im Dokumentationeblock nicht steht und auf die auch nicht aus dem Doku-

mentationsblock heraus verwiesen wird, handelt es sich um die vierte Form. Wenn diese Formen nicht aufgetreten sind, wurde dem API-Element Form 1 zugewiesen.
Die Tabelle 5 zeigt, dass die Java-API-Dokumentation eine „höhere“ Lokalität hat, weil der Anteil, der API-Elemente, die in Form 1 sind bei der Java-API-Dokumentation größer ist, als bei der Python-API-Dokumentation.
Beim „ZipFile-zipfle“-Paar gab es undokumentierte bzw. unverständlich dokumentierte Flags. Diese Flags sind in der Domäne vermutlich Grundwissen, weil auf sie nicht eingegangen wird. Diese Elemente wurden in der Quantifizierung ignoriert, weil sie in keine der vier Formen gepasst haben.

API-Typ	**Form 1**	**Form 2**	**Form 3**	**Form 4**
Calendar	94 (97%)	3 (3%)	0	0
datetime	76 (76%)	14 (14%)	10 (10%)	0
Socket	47 (94%)	2 (4%)	0	0
socket	50 (51%)	31 (32%)	0	17 (18%)
Random	12 (100%)	0	0	0
random	19 (90%)	0	0	2 (10%)
ZipFile*	12 (100%)	0	0	0
zipfile*	31 (86%)	2(6%)	3 (7%)	1 (<1%)
String	51 (72%)	20 (28%)	0	0
string	15 (65%)	5 (21%)	3 (14%)	0

Tabelle 5: Häufigkeit und Verteilung der Formen von Lokalität. Die oberen Paarkomponenten sind API-Typen der Java-API und die unteren der Python-API. *: Es wurden manche API-Elemente ignoriert, siehe Beschreibung.

5.4 Beziehungssichtbarkeit

Beim Entwurf von APIs wird das Prinzip der Modularisierung verwendet. Das führt dazu, dass APIs aus mehreren API-Typen und API-Elementen bestehen, unter denen es Abhängigkeiten gibt. Das ist zum einen für die API-Nutzer von Vorteil, da diese nicht immer alle Funktionalitäten einer API benötigen, sondern nur bestimmte. Die Modularisierung ordnet diese Funktionalitäten in Module und sorgt dafür, dass sie schneller zu finden und leichter zu benutzen sind. Für die Entwickler der API hat ein modularisierter Aufbau einer API den Vorteil, dass es einfacher zu warten und zu erweitern ist. Je nach Modularisierungsgrad kann selbst die Umsetzung simpler Funktionalitäten es erfordern, dass API-Elemente aus mehreren API-Typen benutzt werden müssen. In diesem Fall kann der Dokumentationsstil entscheiden, wie auf diese Beziehungen hingewiesen werden soll.

Java-API Da es sich bei Java um eine statisch und stark typisierte Sprache handelt, kann die Signatur einer Methode bereits viele Informationen enthalten. Bei der Implementierung einer Methode muss z.B. der Typ des Rückgabewerts vor dem Namen platziert werden oder mit `void` angegeben werden, dass es keinen Rückgabewert gibt. Auch der Typ des Eingabewerts muss bereits im Methodenkopf definiert werden, wenn die Methode Eingabeparameter hat. Natürlich kann jedes Mal `Object` als Typ angegeben werden, da alle Klassen in Java der Klasse `Object` untergeordnet sind. Das ist aber in den meisten Fällen nicht zu empfehlen, da dadurch die Vorteile der statischen Typisierung verringert werden. In der `Calendar`-Klasse der Java-API gibt es folgende Methode:

```
public void setTimeZone(TimeZone value)
```

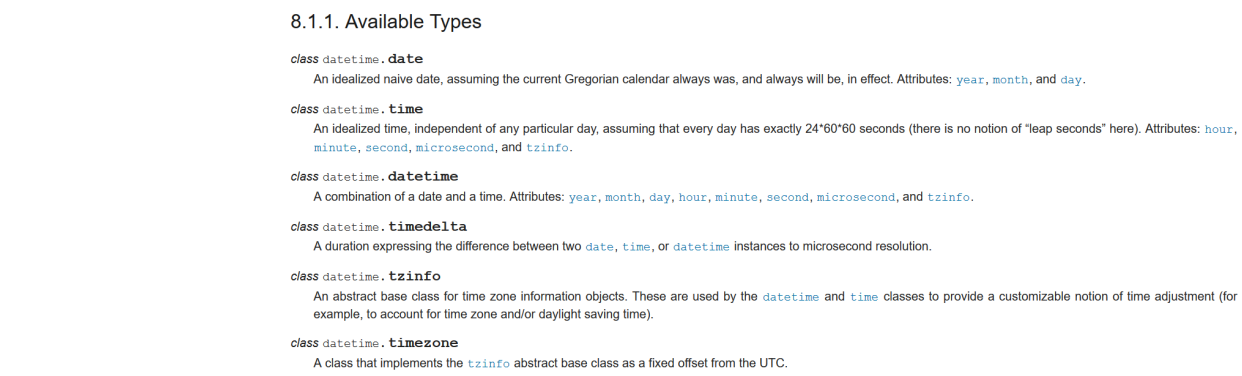
Sets the time zone with the given time zone value.

Parameters:
`value` - the given time zone.

Als Eingabeparameter wird eine Variable des Typs `TimeZone` erwartet. Der Hyperlink auf `TimeZone` führt zu der Klasse `TimeZone` im gleichen Paket `java.util`. Bei einer dynamischen und weniger starken Typisierung wäre es nicht klar, da in der Beschreibung auch kein Hinweis auf die Klasse `TimeZone` existiert. In der Eingangsbeschreibung der Klasse `Calendar` werden die Bezüge zu anderen Klassen, wie z.B. `TimeZone` nicht erklärt. Diese muss der Leser sich selber erarbeiten, denn auch in der Beschreibung des Pakets, dessen Teil `Calendar` und `TimeZone` sind, wird nicht auf die Beziehung dieser beiden Klassen eingegangen. Die einzigen Hinweise sind die Hyperlinks, die in Ein- und Ausgabeparametern von API-Elementen stehen und hin und wieder taucht es in Methodenbeschreibungen auf. Die automatisch generierten „See Also“-Abschnitte sagen oft nichts aus und liefern daher keine wichtigen Informationen.
Ein weiterer Aspekt ist der Grad der Modularisierung. Puristen des objektorientierten Paradigmas neigen zu einer sehr starken Modularisierung. Überspitzt ausgedrückt, ist in Java alles, was eine Klasse sein kann eine Klasse. Das sorgt für sehr viele API-Typen, die aber konzeptuell sehr eng miteinander verwoben sind. Es tauchen in den Konstruktor- und Methodensignaturen der Klasse `Calendar` fünf Mal die Klasse `TimeZone` und sechs Mal die Klasse `Locale` auf. So können komplexe Netze von API-Typen entstehen, die im schlimmsten Fall vom Nutzer selber aufgedeckt werden müssen.

Python-API Anders als bei Java ist es bei Python möglich und üblich mehrere „top-level“-Klassen in einer Quelldatei zu haben. Des Weiteren ist

Pythons Typisierung dynamisch und nicht so stark wie die von Java. Hinzu kommt, dass in der Python-API nicht so stark modularisiert wird. Das drückt sich dadurch aus, dass mehr mit bereits bestehenden Datentypen versucht wird zu arbeiten. Für URLs gibt es in der Java-API eine eigene Klasse. Methoden, die z.B. eine Internetseite als Eingabe haben, erwarten diese als ein Objekt der Klasse `URL`. Bei Python hingegen kann dafür ein String verwendet werden.
Aus diesen Gründen gibt es weniger API-Typen, was zu weniger potenziellen Beziehungen zwischen API-Typen führt. Dadurch, dass in den Signaturen der API-Elementen der Python-API die Typen für die Ein- und Ausgabewerte nicht spezifiziert sind, muss verbal auf solche Beziehungen hingewiesen werden.
Eine Dokumentationsseite der Python-API-Dokumentation ist in mehrere Abschnitte gegliedert, wobei die Abschnitte oft für Klassen bzw. Objekte des Moduls stehen, um das es geht. Es wird, wie in Abbildung 17 zu sehen ist, ganz am Anfang (zu erkennen an der Kapitelnummerierung) auf die Klassen und ihren Wechselwirkungen eingegangen.



8.1.1. Available Types

class datetime.**date**
An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: year, month, and day.

class datetime.**time**
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of "leap seconds" here). Attributes: hour, minute, second, microsecond, and tzinfo.

class datetime.**datetime**
A combination of a date and a time. Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

class datetime.**timedelta**
A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

class datetime.**tzinfo**
An abstract base class for time zone information objects. These are used by the datetime and time classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.**timezone**
A class that implements the tzinfo abstract base class as a fixed offset from the UTC.

Abbildung 17: Die Klassen des Pythonmoduls `datetime` werden inklusive ihren Beziehungen zueinander kurz erklärt.[Quelle: https://docs.python.org/3.4/library/datetime.html(letzter Zugriff: 10.05.2017)]

Diskussion Die Beziehungssichtbarkeit wird von der Typisierung der Sprache, Struktur der API und der API-Dokumentation stark beeinflusst. Wichtige Beziehungen zwischen API-Typen und -Elementen sollten aus der API-Dokumentation hervorgehen, vor allem dann, wenn es wie bei Java sehr viele API-Typen gibt. Wenn diese Beziehungen nicht explizit aufgezeigt und erklärt werden, kann eine statische und starke Typisierung nützlich sein, da dann diese Beziehungen zumindest über die Signaturen der API-Elemente aufgespürt werden können. Die Signaturen werden für die Dokumentation automatisch erstellt. Das könnte die Autoren der Java-API-Dokumentation

dazu verleiten Informationen über Beziehungen versehentlich nicht zu erwähnen, weil diese ja schon in ganz einfacher Form existieren. Das ist bei der Python-API-Dokumentation unwahrscheinlicher, da sich der Autor dort intensiver Gedanken darüber machen muss, welche Informationen er erwähnt und welche nicht.

Quantifizierung Eine gründliche Quantifizierung vom Grad der Abwesenheit von Beschreibungen über Beziehungen ist sehr aufwändig. Zunächst müssten alle Beziehungen der Java- und der Python-API ermittelt werden und anschließend überprüft werden, wie viele dieser Beziehungen implizit und explizit in der Dokumentation beschrieben werden.
Es muss ebenfalls geklärt werden, was genau eine Beziehung ist. Eine Beziehung so zu definieren, dass zwei API-Typen bzw. -Elemente zueinander in Beziehung stehen, wenn sie gemeinsam verwendet werden können, wäre zu extrem. Es gibt nämlich API-Elemente, die so gut wie mit jedem anderen API-Typ kombinierbar sind, wie z.B. die Java-Methode `getClass()`[33]. Wenn alle Beziehungen dieser Art in der Dokumentation erwähnt werden würden, wäre die Dokumentation kaum zu gebrauchen, weil es den Leser mit dieser Art von Informationen überwältigen würde. Außerdem wäre ein Großteil dieser Informationen uninteressant bzw. offensichtlich.
Eine alternative Definition: Zwei API-Elemente stehen zueinander in Beziehung, wenn es intern einen Aufruf untereinander gibt.
Diese Definition würde das Aufspüren von Beziehungen stark vereinfachen, da es Software-Analyse-Werkzeuge gibt, die interne Aufrufe auf Code-Ebene aufspüren können[34]. Diesmal wäre die Definition aber zu eng und viele relevante Beziehungen würden nicht als solche erkannt werden.
Eine Beziehung zwischen zwei API-Elementen bzw. -Typen besteht dann, wenn sie gemeinsam genutzt werden können, um eine Funktionalität so zu erreichen, wie sie von den Entwicklern der API bewusst konzipiert wurde. Diese Definition scheint am ehesten die Beziehungen zu meinen, auf die in der Dokumentation hingewiesen werden sollte. Es ist aber nicht sehr leicht zu entscheiden, ob eine Beziehung diese Kriterien erfüllt oder nicht.

5.5 Wissenstypenordnung

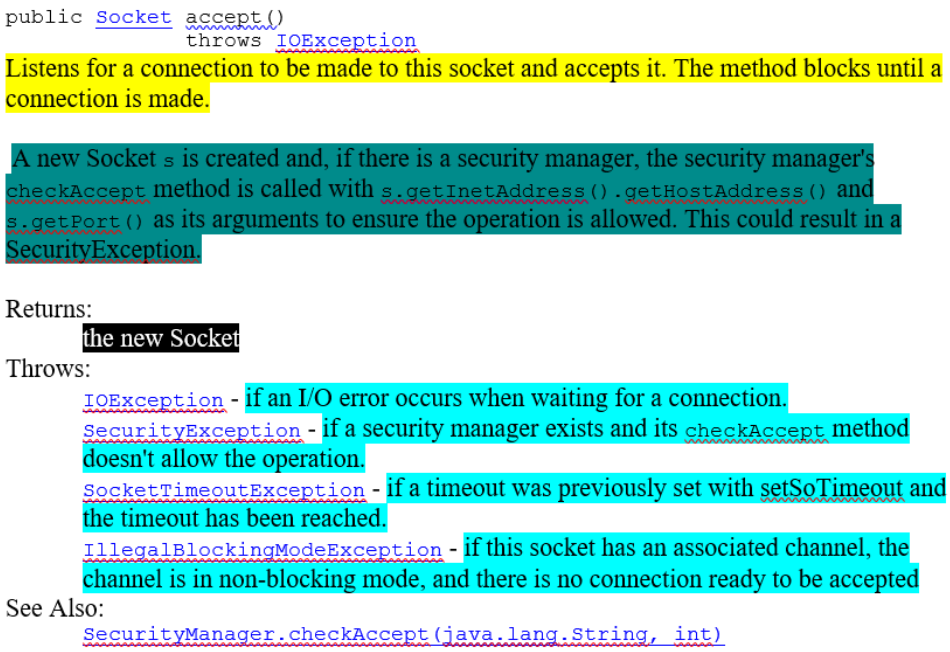
Ein weiterer großer Einflussfaktor, der den Stil der Dokumentation ausmacht, ist die natürliche Sprache. Denn den wichtigsten Anteil einer API-Dokumentation machen die Beschreibungen und Informationen aus, die die Autoren schreiben. Zu den vielen Entscheidungen, die die Autoren treffen müssen gehören z.B. welche Informationen in eine Beschreibung aufgenom-

[33] https://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#getClass() (letzter Zugriff: 10.05.2017)
[34] https://softvis.wordpress.com/tools/ (letzter Zugriff: 10.05.2017)

men werden sollen, wie detailliert auf etwas eingegangen werden soll und in welcher Reihenfolge auf bestimmte Aspekte eingegangen werden soll.
Um besser über diese Themen sprechen zu können und um den Inhalt von API-Dokumentationen besser analysieren zu können, haben Maalej und Robillard in ihrer Studie [1] (siehe Kapitel 3.3) eine Taxonomie von Wissenstypen erstellt. Diese Wissenstypen geben an, welche Art von Information ein Textabschnitt aus der API-Dokumentation liefert. Wie in Kapitel 4.1 beschrieben, habe ich zwei Paare von Dokumentationsseiten erstellt und dann Satz für Satz entschieden, welchen Wissenstyp dieser Satz enthält. Dabei ist mir aufgefallen, dass es bei der Typisierung der Dokumentation der Python-Module `datetime` und `socket` häufiger einen Wechsel des Wissenstyps gab.
Dieser Unterschied könnte daher kommen, dass es in der Anleitung für Autoren der Java-API-Dokumentation Listen darüber gibt, welche Informationen in einer Paket-, Klassen-, Interface-, Methoden- und Feldbeschreibung stehen sollen. Es steht zwar nicht, dass die Reihenfolge der Liste eingehalten werden muss, ist aber wahrscheinlich meistens der Fall. In der Anleitung für Autoren der Python-API-Dokumentation sind solche Vorgaben viel schwächer und grober.

Java-API In Java sind die Wissenstypen „aufgeräumter“. Das heißt, dass es seltener vorkommt, dass in einem Satz mehrere Wissenstypen enthalten sind. Wie Abbildung 18 zeigt, sind die Wissenstypen in der Beschreibung der Methode `accept()` hauptsächlich in drei Abschnitte unterteilt.
Natürlich kommt es des Öfteren zu einer stärkeren Durchmischung von Wissenstypen, aber dieser Trend der geordneten Wissenstypverteilung scheint mir in Java konzeptuell zu geschehen und je mehr ich mich damit beschäftigt habe, desto mehr wurde ich mir dessen sicher.

Python-API Im Gegensatz zur Java-API-Dokumentation, sind die Wissenstypen in der Python-API-Dokumentation etwas stärker durchmischt. Es ist also üblicher in einem Satz auf unterschiedliche Wissenstypen oder auf konsekutive Sätze in unterschiedlichen Wissenstypen zu treffen.
Abbildung 19 zeigt die Typisierung der Methode `fromtimestamp`(*timestamp, tz=None*) aus dem Modul `datetime`. Es wird öfter zwischen unterschiedlichen Wissenstypen hin und her gewechselt.
Dieser Trend ist vermutlich dem Umstand zuzuschreiben, dass die Python-API-Dokumentation dem Autor viel weniger vorschreibt, wie eine Dokumentationsseite auszusehen hat und wo was stehen muss. Dadurch kann der Autor jene Informationen aufnehmen, die er für wichtig hält und die Reihenfolge so gestalten, wie er es für am besten hält.



```
public Socket accept()
              throws IOException
```

Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

A new Socket `s` is created and, if there is a security manager, the security manager's `checkAccept` method is called with `s.getInetAddress().getHostAddress()` and `s.getPort()` as its arguments to ensure the operation is allowed. This could result in a SecurityException.

Returns:

the new Socket

Throws:

`IOException` - if an I/O error occurs when waiting for a connection.

`SecurityException` - if a security manager exists and its `checkAccept` method doesn't allow the operation.

`SocketTimeoutException` - if a timeout was previously set with setSoTimeout and the timeout has been reached.

`IllegalBlockingModeException` - if this socket has an associated channel, the channel is in non-blocking mode, and there is no connection ready to be accepted

See Also:

`SecurityManager.checkAccept(java.lang.String, int)`

Abbildung 18: Wissenstypen der Beschreibung der Java-Methode `accept()` der Klasse `Socket`. Farblegende: Functionality and Behaviour Control-Flow Non-Information Directives

Diskussion Eine geordnete Wissenstypverteilung hat den Vorteil, dass der Leser schneller seine spezifisch gesuchte Information finden kann. Wenn ein größerer Textblock nur einen Wissenstyp enthält und der Leser sich für diesen Wissenstyp nicht interessiert, kann er diesen Block überspringen, ohne befürchten zu müssen, dass die gesuchte Information in diesem Textblock war. Diese Art der geordneten Wissenstypverteilung bringt aber den Nachteil mit sich, dass er für den Leser langweiliger und trockener sein kann, weil ein Thema zu intensiv behandelt wird und eventuell Informationen auftauchen, die den Leser an dieser Stelle nicht interessieren. Das kann Leser stärker ermüden, die ein vollständiges Verständnis über die API haben wollen und deshalb die gesamte Dokumentation wie ein Buch lesen möchten.
Für solche Leser ist eine etwas lockerere Form der Wissenstypverteilung vermutlich angenehmer. Einerseits können die häufigen Wechsel der Wissenstypen irritierend sein, weil zu oft das Thema gewechselt wird, andererseits kann es lesbarer sein, weil der Text „natürliche“ Gedankengänge widerspiegeln kann. Dadurch kann ein besserer Lesefluss entstehen, wodurch dem Leser auf eine didaktisch geschickte Art die API beigebracht werden kann, ohne ihn zu langweilen.

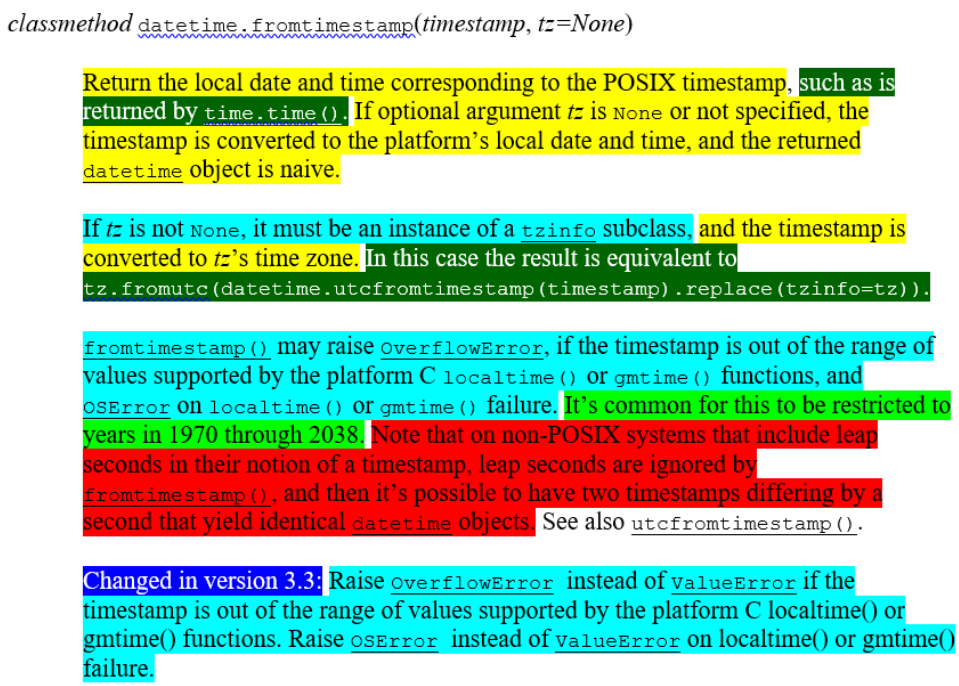


Abbildung 19: Wissenstypen der Beschreibung der Python-Methode `fromtimestamp`*(timestamp, tz=None)* aus dem `datetime`-Modul. Farblegende: Functionality and Behaviour Structure and Relationship Concepts Directives Quality Attributes and Internal Aspects Environment

Quantifizierung Eine gründliche Quantifizierung erfordert die vollständige Typisierung von vielen Dokumentationsseiten. Dann müsste überprüft werden wie viele Textblöcke und Sätze es gibt, wie oft in einem Textblock bzw. in einem Satz der Wissenstyp gewechselt wird und wie viele verschiedene Wissenstypen es in einem Satz gibt. Interessant wären auch wiederkehrende Muster bzw. Abfolgen von Wissenstypwechseln zu ermitteln.
Die von mir typisierten Paare eignen sich nicht so gut, da sie künstlich zusammengestellt wurden und sie die ursprünglichen Strukturen der Dokumentationsseiten nicht mehr haben. Dennoch habe ich gezählt, wie oft in der Beschreibung der API-Elemente der Wissenstyp in einem Block und in einem Satz gewechselt wurde.
Pendant 1: datetime(Python) - Date, Calendar, GregorianCalendar, SimpleDateFormat (Java)
Im Python-Teil des Pendants gibt es 22 API-Elemente und die Länge der Beschreibungen beträgt ohne Leerzeichen 12823 Zeichen. 83 Mal gibt es einen Wechsel des Wissenstyps und das heißt, dass es im Durchschnitt etwa alle 154 Zeichen zu einem Wechsel kommt.

Im Java-Teil des Pendants gibt es 25 API-Elemente, deren Beschreibungen ohne die Leerzeichen eine Gesamtlänge von 12450 Zeichen haben. Das ergibt im Durchschnitt etwa einen Wechsel alle 53 Zeichen.
Pendant 2: socket(Python) - Socket, ServerSocket, InetSocketAddress, SocketChannel, SocketImpl (Java)
Im Python-Teil des Pendants gibt es 13 API-Elemente, die ohne Leerzeichen mit 5461 Zeichen beschrieben und erklärt sind. Dabei gibt es an 55 Stellen einen Wechsel zwischen den Wissenstypen, sodass es ca. alle 99 Zeichen zu einem Wechsel kommt.
Im Java-Teil des Pendants sind 15 Elemente. Die Beschreibung ist 7420 (wieder ohne Leerzeichen) Zeichen lang und es gibt 39 Wechsel der Wissenstypen, was durchschnittlich auf einen Wechsel alle 162 Zeichen führt.
Dass es auf der Java-Seite mehr API-Elemente gibt liegt daran, dass einige API-Elemente auf der Python-Seite nur mit einer Kombination aus mehreren API-Elementen aus der Java-Seite nachzubilden sind.
Aufgrund der sehr niedrigen Anzahl der betrachteten Pendants und der Veränderung der Dokumentationsseiten, bedarf es einer ausführlicheren Quantifizierung mit viel mehr Dokumentationsseiten, um dieses Phänomen überzeugend quantifiziert zu haben.

5.6 Zusammenfassung

In der Vorstellung der Unterschiede zwischen der Java- und Python-API-Dokumentation fällt häufig auf, dass diese Dokumentationen zwei unterschiedliche Stile haben.
Es folgt eine Vorstellung des *Referenzstils* und des *Buchstils*, in der die Stile gegenübergestellt werden.

5.6.1 Referenzstil

Aufbau und Form Der Aufbau und die Form der Dokumentation ähneln stark dem der API. Sie ist modular, sodass jeder API-Typ seine eigene Dokumentationsseite hat und die Dokumentationsseiten listen alle API-Elemente auf und jedes API-Element wird in seinem eigenen Dokumentationsblock beschrieben. Die Klassenbeschreibung dient als eine Art Einleitung und erklärt die Funktionsweise des API-Typs und die wichtigen Konzepte.

5.6.2 Buchstil

Aufbau und Form Der Aufbau und die Form der Dokumentation ähnelt stark einem Buch. Sie ist in Kapitel aufgeteilt, sodass die Dokumentationsseiten thematisch in Kapiteln gruppiert sind und selber auch Kapitel enthalten, die die API-Elemente erklären. Die Reihenfolge der der Dokumentationsseiten ist also nicht alphabetisch, sondern unterliegt didaktischen Überlegungen. Die früheren Kapitel enthalten wichtigere und grundlegende Informationen,

Vorteile:
Ein Leser, der sich mit dem Aufbau der API gut auskennt und weiß, nach welchem API-Typ er suchen muss, findet schneller die richtige Dokumentationsseite.
Ein Leser, der mit der Form der Dokumentationsseite vertraut ist, findet schneller die passende Stelle.

Nachteile:
Für Leser, die sich mit dem Aufbau der API nicht gut auskennen, nicht wissen, von welchem API-Typ sie die Dokumentation brauchen und/oder sich mit der Form der Dokumentationsseiten nicht auskennen, gilt das Gegenteil der oben genannten Vorteile.

Beispielnutzung In den Text eingebettete Code-Beispielschnipsel werden in der Dokumentation dafür eingesetzt, um die Elemente zu erklären bzw. die Funktionalität zu verdeutlichen. Sie werden nicht eingesetzt, um die Umsetzung bestimmter Anwendungsfälle oder „Best Practices“ zu beschreiben. Die sind in Tutorials/Guides platziert, die in der API-Dokumentation an den relevanten Stellen verlinkt sind.

Vorteile:
Durch das Auslagern großer Code-Beispielblöcke in Tutorials/Guides wirkt die Dokumentationsseite auf-

die späteren eher spezielle.

Vorteile:
Ein Leser, der die API noch nicht gut kennt und die Dokumentation dazu nutzen möchte, um die API zu lernen, wird besser orientiert. Die Reihenfolge und Namen der Kapitel geben ihm Auskunft darüber, ob er an der richtigen Stelle ist.
Die Dokumentationsseite ist so gestaltet, dass der Leser die API möglichst einfach gründlich versteht. Es wird dabei empfohlen, dass er die Dokumentation wie ein Buch liest.
Nachteile:
Für Leser, die die API bereits kennen ist es etwas schwieriger sich schnell in der Dokumentation zu bewegen, als beim Referenzstil

Beispielnutzung Ähnlich wie beim Referenzstil wird auch im Buchstil Gebrauch von eingebetteten Code-Beispielschnipseln gemacht, um die Elemente zu erklären bzw. die Funktionalität zu verdeutlichen. Zusätzlich werden in größeren Code-Beispielblöcken Anwendungsfälle und „Best-Practices“ erklärt. Es kann trotzdem Tutorials/Guides geben, die in der API-Dokumentation an den relevanten Stellen verlinkt sind, die die Beispiele detaillierter erklären.
Vorteile:
Durch die Verwendung von größeren Code-Beispielblöcken können verbal beschriebene Sachverhalte in die

geräumter. Code-Beispielschnipsel können in die Beschreibungen eingebettet werden, um die Funktionalität der API-Elemente besser erklären zu können, ohne der Form und der Übersichtlichkeit zu schaden.	Praxis umgesetzt und illustriert werden. Die Leser können die Code-Beispielblöcke in ihren Quellcode einbinden und bekommen so von der API-Dokumentation Lösungen. Code-Beispielschnipsel können in die Beschreibungen eingebettet werden, um die Funktionalität der API-Elemente besser erklären zu können.
Nachteile: Gerade Leser, die nicht daran interessiert sind viel Dokumentation zu lesen und die sonst viel vom Referenzstil profitieren, werden ausgebremst. Denn sie lernen oft anhand von Beispielen, wie Studien gezeigt haben (siehe Kapitel 3). Wenn es keine Tutorials/Guides gibt, die die API-Elemente abdecken, fehlen dem Nutzer die Beispiele.	*Nachteile:* Durch größere Code-Beispielblöcke wächst die Dokumentationsseite, was die Navigation erschwert und der Übersichtlichkeit schadet. API-Dokumentationen mit größeren Code-Beispielblöcken sind schwerer und aufwändiger zu pflegen, da eine Änderung der API dazu führen kann, dass die Code-Beispielblöcke nicht mehr korrekt sind und angepasst werden müssen. Dem Leser kann die falsche Erwartung auferlegt werden, dass alle API-Elemente in den Code-Beispielblöcken gezeigt werden. Der Leser kann somit nur noch die Code-Beispielblöcke betrachten und, wenn ein API-Element, das er gebraucht hätte nicht im Beispiel vorkommt, könnte er seine Suche fälschlicherweise woanders fortsetzen.
Lokalität Die Informationen über ein Element sind nicht über die Dokumentationsseite verteilt, sondern sind kompakt an einem Ort. Wird stark vom Aufbau und von der Form der Dokumentationsseite beeinflusst.	**Lokalität** Die Informationen über ein Element dürfen stärker über die Dokumentationsseite verteilt werden, sodass ein Element aus unterschiedlichen Betrachtungswinkeln erklärt werden kann. Wird stark vom Aufbau und der Form der Dokumentationsseite beeinflusst.
Vorteile: Die Dokumentation wirkt aufgeräumter und die relevanten Informa-	*Vorteile:* Für Autoren ist es leichter auf Zusammenhänge zwischen API-Elementen

tionen über ein API-Element können schneller zusammengetragen werden.
Nachteile:
Für Autoren ist es schwerer auf Zusammenhänge zwischen API-Elementen hinzuweisen bzw. diese zu beschreiben.
Dadurch müssen die Leser diese Informationen sich selber herleiten.

Beziehungssichtbarkeit Aufgrund des Aufbaus und der Form der API-Dokumentation, muss in den Dokumentationsseiten stärker auf die Beziehungen zwischen API-Typen bzw. -Elementen eingegangen werden. Darauf kann teilweise verzichtet werden, wenn die API stark modularisiert und typisiert ist. Denn dadurch können viele wichtige Beziehungen über die Signaturen der API-Elemente ermittelt werden.

Vorteile:
Es gibt weniger Text und nicht immer ist eine Beziehung relevant für einen Leser.
Die API-Elemente können vom Leser isolierter betrachtet werden, was das Verständnis für das API-Element steigern kann.

Nachteile:
Die Beziehungen müssen vom Leser selber ermittelt werden und es kann dazu führen, dass der Nutzer denkt, dass bestimmte Aufgaben mit der API nicht zu lösen sind, weil dazu API-Typen kombiniert werden müssen, der Leser das aber nicht erfährt.

hinzuweisen bzw. diese zu beschreiben.

Nachteile:
Die Dokumentation wirkt unordentlicher und es kann mühsam sein alle relevanten Informationen über ein API-Element zusammenzutragen, weil die Dokumentationsseite intensiver gelesen werden muss.

Beziehungssichtbarkeit Da die Dokumentation in Kapitel unterteilt sind und mehrere, zueinander in Beziehung stehende API-Typen in der selben Dokumentationsseite sind, ist eine höhere Beziehungssichtbarkeit gegeben. Das legt den Autoren nahe stärker über die Beziehungen zwischen den API-Typen und -Elementen zu berichten. Wenn die API nicht so stark modularisiert und typisiert ist, wird das aber zwingend notwendig, da es sonst sehr schwer wird auf Beziehungen aufmerksam zu werden.
Vorteile:
Eine hohe Beziehungssichtbarkeit kann den Leser auf ähnliche, unterstützende und ergänzende API-Typen und -Elemente aufmerksam machen, die für die Umsetzung bestimmter Ziele von Bedeutung sein können.
Außerdem kann der Leser den Aufbau der API besser nachvollziehen.
Nachteile:
Bei sehr vielen Beziehungen kann eine Beschreibung jeder Beziehung zu unübersichtlichen und langen Dokumentationsseiten führen.

Wissenstypenordnung Textabschnitte enthalten möglichst nur einen Wissenstyp und sind gut voneinander zu unterscheiden. Sprünge zwischen Wissenstypen werden eher gemieden. Es gibt eine feste Reihenfolge von Wissenstypen.

Vorteile:
Die Autoren können mit einer Checkliste die Wissenstypen abarbeiten und bekommen eine Hilfe bei Strukturieren ihres Textes.
Die Leser können den Text schneller überfliegen, weil sie zu Beginn des Textabschnitts schon entscheiden können, ob es sich für sie lohnt diesen Textabschnitt weiterzulesen oder zum nächsten überzugehen.

Nachteile:
Der Text kann länger werden, da unter Umständen mehr Redundanz entsteht, weil bestimmte Informationen unterschiedlicher Wissenstypen, die eigentlich im gleichen Satz erwähnt werden könnten, in unterschiedlichen Sätzen erwähnt werden müssen.
Wenn der Leser den ganzen Text lesen möchte oder muss, kann es ihn langweilen einen Abschnitt nur über einen Wissenstyp zu lesen, weil er frühzeitig Informationen erhält, die ihn erst zu einem späteren Zeitpunkt interessieren würden.

Wissenstypenordnung Einer geordneten und strukturierten Wissenstypenordnung wird nicht so viel Beachtung gegeben. Den Leser möglichst effektiv die wichtigen Informationen zu vermitteln und den Dokumentationstext so zu gestalten, dass es möglichst einfach zu verstehen ist, ist im Buchstil wichtiger.
Vorteile:
Die Autoren sind freier in der Gestaltung des Textes und können deswegen je nach Thema entscheiden wie sie es dem Leser vermitteln wollen.
Die Autoren können einen besseren Lesefluss durch die Unterstützung von „natürlichen“ Gedankengängen ermöglichen.
Die Dokumentation kann verkürzt werden, da verschachtelte Sätze kürzer sein können, weil weniger Redundanz herrscht. Das bedeutet, dass in einem Nebensatz ein anderer Wissenstyp sein kann, als in einem Hauptsatz, anstatt die Information im Nebensatz drei Sätze später anzubringen.
Nachteile: Leser, die die Dokumentation überfliegen möchten, anstatt sie gründlich zu lesen und zu verstehen werden bei einer ungeordneten Wissenstypverteilung ausgebremst. Denn sie können nicht so schnell entscheiden, ob sie einen Textabschnitt überspringen können, oder nicht.
Dadurch, dass Autoren mehr Freiheiten in der Gestaltung des Textes haben, können sie aber auch schlechtere Dokumentation erzeugen, weil sie weniger Hilfen haben. Denn eine bewährte, fest vorgegebene Reihenfolge kann für viele Autoren besser sein, als, dass sie die Reihenfolge selber bestimmen müssten.

5.6.3 Referenzstil vs. Buchstil

Durch die unterschiedlichen Ausprägungen der Stileigenschaften, wirken API-Dokumentationen im Referenzstil geordneter und aufgeräumter, als API-Dokumentationen im Buchstil. Das führt dazu, dass Leser, die genau wissen wonach sie suchen müssen, schneller an ihr Ziel geführt werden können. Im Gegensatz zum Buchstil ist der Aufbau der Dokumentation und die Form der Dokumentationsseiten weniger semantisch, sondern mehr förmlich. Das führt dazu, dass es Leser, die sich mit der API bzw. der API-Dokumentation noch nicht so gut auskennen, schwerer haben sich in der Dokumentation zurechtzufinden.
Diese werden vermutlich in API-Dokumentationen, die den Buchstil haben besser zurecht kommen. Denn anders als im Referenzstil, sind die API-Dokumentationen im Buchstil so konzipiert, dass sie wie ein Buch von Anfang an gelesen werden sollen, möglichst ohne etwas zu überspringen. Denn es werden Begriffe und Konzepte eingeführt, die einmal definiert und erklärt werden und später an mehreren Stellen verwendet werden. Damit der Leser sich nicht langweilt und der Text sich gut lesen lässt, ist ein guter Lesefluss wichtig. Leser, die sich mit der API, der API-Dokumentation und den meisten Konzepten der Domänen auskennen, werden vom Buchstil ausgebremst. Denn für sie ist es aufwändiger den Text nach für sie wichtige Informationen durchzukämmen.
Somit ist der Referenzstil eher als ein Nachschlagewerk geeignet und der Buchstil eher als eine Gebrauchsanweisung.

6 Fazit

In diesem Kapitel werden die Inhalte der vorherigen Kapitel, sowie die Ergebnisse der vorliegenden Masterarbeit zusammengefasst. Dabei wird auch Stellung zu den Zielen genommen, die in der Einleitung (siehe Kapitel 1) für die Masterarbeit gesetzt wurden.
Das Kapitel und somit die Masterarbeit wird mit einem Ausblick beendet.

6.1 Zusammenfassung

Softwareentwicklung ohne den Einsatz von APIs ist für viele Programmierer heutzutage nicht mehr vorstellbar. Denn sie bieten vorgefertigte Lösungen an, die in die eigene Software eingebunden werden können. Das ist in der Regel viel bequemer und zeitsparender, als alles von Grund auf selber zu programmieren und bietet weitere Vorteile, wie in Kapitel 2 dargelegt wird.
Genauso wie es eine Qual sein kann Programme mit schlechter Benutzerfreundlichkeit zu benutzen, kann eine schlechte Benutzerfreundlichkeit auch den Einsatz einer API erheblich erschweren.
Wie mehrere Studien herausgefunden haben (siehe Kapitel 3), spielt die API-Usability eine große Rolle dabei, wie gut die API eingesetzt werden kann und dass sich eine schlechte API-Usability schlecht auf das Produkt auswirken kann, bei dessen Erstellung die API genutzt wurde.
Weitere Studien haben einen oft negativen Einfluss der API-Dokumentation auf die Benutzerfreundlichkeit einer API herausgefunden.
Das macht die Erforschung von API-Dokumentationen wichtig, denn sie können eine Verbesserung der API-Dokumentationen bewirken.
Vorgefertigte Designentscheidungen, die von Autoren von API-Dokumentationen dazu verwendet werden können je nach Szenario ihre Dokumentation zu entwerfen und zu schreiben, sind erstrebenswert.
Das übergeordnete Ziel dieser Masterarbeit ist es, diesem Ziel näher zu kommen. Dazu wurde die Hypothese überprüft, dass es unterschiedliche Dokumentationsstile gibt. Die Idee ist, dass unterschiedliche Dokumentationsstile verschiedene Vor- und Nachteile haben und entsprechend für unterschiedliche Szenarien besser oder schlechter geeignet sind.
Zu diesem Zweck wurden die von Maalej und Robillard vorgestellten [1] und von Wildermann erweiterten [34] Wissenstypen dazu verwendet Teile der Java- und der Python-API-Dokumentation zu typisieren. Das hat sich als eine gute Methode herausgestellt, um unterschiedliche Dokumentationsstile und Stileigenschaften von API-Dokumentationen zu erkennen. Dies war nämlich ein weiteres Ziel der Masterarbeit. Mit diesen Stileigenschaften kann nämlich untersucht werden, ob unterschiedliche API-Dokumentationen unterschiedliche Stile haben. Die Untersuchung, deren genauere Details Kapitel 4 zu entnehmen sind, ergab, dass es fünf Stileigenschaften gibt, die Stile von API-Dokumentationen beschreiben. Es besteht jedoch kein An-

spruch auf eine Vollständigkeit. Denn es ist durchaus möglich, dass es noch weitere Stileigenschaften gibt. Die bisher entdeckten Stileigenschaften beziehen sich sowohl auf die Form und Präsentation von API-Dokumentationen, als auch auf den Inhalt, wie in Kapitel 5 erläutert wird. Außerdem werden in Kapitel 5 ausgehend von den Stileigenschaften zwei Dokumentationsstile vorgestellt, deren Aufspüren und Beschreibung ebenfalls ein gesetztes Ziel dieser Masterarbeit war. Die Dokumentationsstile konnten anhand der unterschiedlichen Ausprägungen der Stileigenschaften in der Java- und der Python-API-Dokumentation ermittelt werden.
Neben der Überprüfung der Hypothese, dass es unterschiedliche Stile von API-Dokumentationen gibt, der Identifizierung von Stileigenschaften und Dokumentationsstilen, sind die Erprobung und Entwicklung von Methoden, sowie ein Zusammentragen wichtiger Erkenntnisse für die Forschung in diesem Gebiet weitere Ergebnisse dieser Masterarbeit.
Auch wenn es noch offene Fragen gibt, sind die Ziele der Masterarbeit weitgehend erfüllt worden.

6.2 Ausblick

Es wurde zwar geklärt, dass es unterschiedliche Dokumentationsstile gibt, doch sind die Stileigenschaften erstens noch nicht genug ausgereift und zweitens noch nicht in ausreichender Menge aufgedeckt, um daraus Entwurfsmuster für API-Dokumentationen ableiten zu können.
Dazu müssen die vorgestellten Stileigenschaften weiter untersucht und intensiver quantifiziert werden. Wie die Quantifizierung jeweils aussehen muss, ist in Kapitel 5 beschrieben und teilweise vorgeführt.
Es gibt sehr wahrscheinlich noch weitere Stileigenschaften, die noch entdeckt werden müssen. Zum Beispiel habe ich den Verdacht, dass das Abstraktionsniveau der API-Dokumentation, also wie stark in der Beschreibung auf die Implementierung eingegangen wird, ebenfalls eine Stileigenschaft ist, die aber noch weiter erforscht werden muss. Dazu können neben der Java- und Python-API-Dokumentation auch andere API-Dokumentationen betrachtet werden. Eine intensivere Beschäftigung mit Wissenstypen kann ebenfalls positive Ergebnisse bringen. Denn die in großen Mengen vorhandenen typisierten Dokumentationseinheiten, die in den Studien von Maalej, Robillard und Wildermann [1, 34] erstellt wurden, sind lückenhaft und stark verteilt. Es wurden nämlich nicht gesamte Dokumentationsseiten typisiert, sondern Dokumentationseinheiten, die zufällig aus unterschiedlichen Dokumentationsseiten entnommen wurden. Die Typisierung von gesamten Dokumentationsseiten ist jedoch besser geeignet, um Stileigenschaften und Dokumentationsstile zu entdecken.
Um die in dieser Masterarbeit entdeckten Dokumentationsstile zu validieren und diese eventuell weiter auszubauen, können andere API-Dokumentationen analysiert und die Ausprägung der Stileigenschaften überprüft werden.

Somit können erstens die Stileigenschaften darauf überprüft werden, ob sie wirklich allgemeingültig sind und zweitens die Dokumentationsstile anderer API-Dokumentationen untersucht werden.
Dadurch können vorhandene Stileigenschaften und Dokumentationsstile präziser beschrieben und umrissen werden, sowie weitere Stileigenschaften und Dokumentationsstile entdeckt werden.
Bei der Beschreibung der Stileigenschaften in Kapitel 5 wird auch über mögliche Gründe für gewisse Ausprägungen dieser Eigenschaften in den untersuchten Dokumentationen spekuliert. Interessant wäre zu wissen, ob diese Gründe für die Autoren der untersuchten API-Dokumentationen eine Rolle gespielt haben. Es könnte durchaus sein, dass viele Entscheidungen gar nicht so fundiert getroffen wurden, wie die Stileigenschaften vermuten lassen können. Wenn das so ist, wäre es interessant zu erfahren, ob und wie gewisse Entscheidungen getroffen werden und, ob die Abwägungen, die die Stileigenschaften suggerieren tatsächlich gemacht werden.
Sollten diese Methoden keine fruchtbaren Ergebnisse liefern, kann in anderen Wissenschaftszweigen geschaut werden. Sprach- und Textforschung könnten sich dafür eignen, da die Art und Weise, wie die API-Dokumentationen geschrieben sind ebenfalls sehr wichtig für den Stil einer API-Dokumentation ist.
Vermutlich können die Erziehungswissenschaften herangezogen werden, Strategien zu erkennen, die in den API-Dokumentationen verwendet werden, um dem Leser die API beizubringen.
Lernstrategien und Verhaltensweisen von API-Nutzern, die eine API lernen wollen, können mit Forschungen aus der Psychologie untersucht werden.
Insofern ist zu hoffen, dass in der Zukunft weitere Stileigenschaften und Dokumentationsstile gefunden werden und somit dem Ziel näher gekommen wird Entwurfsmuster für API-Dokumentationen aufzustellen.

Literatur

[1] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[2] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, pp. 131–183, June 1992.

[3] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[4] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Commun. ACM*, vol. 39, pp. 104–116, Oct. 1996.

[5] J. M. Daughtry, U. Farooq, B. A. Myers, and J. Stylos, "Api usability: Report on special interest group at chi," *SIGSOFT Softw. Eng. Notes*, vol. 34, pp. 27–29, July 2009.

[6] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "Sometimes you need to see through walls: A field study of application programming interfaces," in *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, (New York, NY, USA), pp. 63–71, ACM, 2004.

[7] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 5–14, Oct 2013.

[8] T. Grechenig, *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten.* IT Informatik, Pearson Studium, 2010.

[9] W. B. Frakes and C. J. Fox, "Sixteen questions about software reuse," *Commun. ACM*, vol. 38, pp. 75–ff., June 1995.

[10] J. Stylos, *Making Apis More Usable with Improved Api Designs, Documentation and Tools.* PhD thesis, Pittsburgh, PA, USA, 2009.

[11] J. Sushine, J. D. Herbsleb, and J. Aldrich, "Searching the state space: A qualitative study of api protocol usability," in *2015 IEEE 23rd International Conference on Program Comprehension*, pp. 82–93, May 2015.

[12] T. Grill, O. Polacek, and M. Tscheligi, *Methods towards API Usability: A Structural Analysis of Usability Problem Categories*, pp. 164–180. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[13] M. F. Zibran, F. Z. Eishita, and C. K. Roy, "Useful, but usable? factors affecting the usability of apis," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pp. 151–155, IEEE, 2011.

[14] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Softw. Engg.*, vol. 16, pp. 703–732, Dec. 2011.

[15] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, (New York, NY, USA), pp. 506–507, ACM, 2006.

[16] E. Berglund, "Designing electronic reference documentation for software component libraries," *J. Syst. Softw.*, vol. 68, pp. 65–75, Oct. 2003.

[17] B. Kahlert, *API-Usability der auf Templatemetaprogrammierung basierenden Softwarebibliothek „SeqAn"*. PhD thesis, Freie Universtität Berlin, 2015.

[18] C. Bore and S. Bore, "Profiling software api usability for consumer electronics," in *2005 Digest of Technical Papers. International Conference on Consumer Electronics, 2005. ICCE.*, pp. 155–156, Jan 2005.

[19] J. Stylos and B. Myers, "Mapping the space of api design decisions," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pp. 50–60, Sept 2007.

[20] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Commun. ACM*, vol. 40, pp. 32–38, Oct. 1997.

[21] M. B. Rosson and J. M. Carroll, "The reuse of uses in smalltalk programming," *ACM Trans. Comput.-Hum. Interact.*, vol. 3, pp. 219–253, Sept. 1996.

[22] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck, "A debugging perspective on end-user mashup programming," in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 149–156, Sept 2010.

[23] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building more usable apis," *IEEE Softw.*, vol. 15, pp. 78–86, May 1998.

[24] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar apis: An exploratory study," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, (Piscataway, NJ, USA), pp. 266–276, IEEE Press, 2012.

[25] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in api design: A usability evaluation," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 302–312, IEEE Computer Society, 2007.

[26] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 529–539, IEEE Computer Society, 2007.

[27] J. Stylos and B. A. Myers, "The implications of method placement on api learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, (New York, NY, USA), pp. 105–112, ACM, 2008.

[28] M. F. Zibran, "What makes APIs difficult to use?," *International Journal of Computer Science and Network Security*, vol. 8, no. 4, pp. 255–261, 2008.

[29] J. Bayer and D. Muthig, "A view-based approach for improving software documentation practices," in *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, pp. 10 pp.–278, March 2006.

[30] C. J. Stettina and W. Heijstek, "Necessary and neglected?: An empirical study of internal documentation in agile software development teams," in *Proceedings of the 29th ACM International Conference on Design of Communication*, SIGDOC '11, (New York, NY, USA), pp. 159–166, ACM, 2011.

[31] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, pp. 68–75, July 2015.

[32] J. A. Jacko, *Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, Third Edition*. Boca Raton, FL, USA: CRC Press, Inc., 3rd ed., 2012.

[33] K. Neuendorf, *The Content Analysis Guidebook*. SAGE Publications, 2002.

[34] S. Wildermann, "Messung der Informationstypen-Häufigkeiten in der Python-Dokumentation," Bachelor's thesis, Freie Universität Berlin, 2014.

[35] P. Mayring, *Einführung in die qualitative Sozialforschung: eine Anleitung zu qualitativem Denken*. Beltz Studium : Erziehung und Bildung, Beltz, 2002.

[36] S. Salinger, *Ein Rahmenwerk für die qualitative Analyse der Paarprogrammierung.* PhD thesis, Freie Universtität Berlin, 2013.

[37] B. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Observations (Chicago, Ill.), Aldine Publishing Company, 1967.

[38] A. Strauss and J. Corbin, *Basics of qualitative research: grounded theory procedures and techniques.* Sage Publications, 1990.

A Anhang

A.1 Ergebnisse der Studie von Zibran et al. (2011)

Abbildung 2 zeigt die Verteilung der Fehlerberichte bezüglich ihrer Thematik.
Etwa 43% der Fehlerberichte berichten eigentlich über keine Fehler, sondern fordern weitere Features. Danach kommen mit circa 32% „echte“ Fehlerberichte, also Fälle, in denen die API sich nicht so verhält, wie es von der Dokumentation versprochen wurde und wie es richtig sein müsste. Das beeinflusst aber mehr die Brauchbarkeit als die Benutzerfreundlichkeit. Denn bei einer unbrauchbaren API, indem Sinne, dass sie nicht funktioniert, erübrigt sich die Benutzerfreundlichkeit.
Ungefähr ein Viertel (∼27%) melden Fehler in der Dokumentation. Neben vielen textuellen Fehlern (Tippfehler, grammatikalische Fehler) gibt es auch weitere Fehler, die auf eine falsche Beschreibung eines Elements oder auf falsche, nicht funktionierende Code-Beispiele basieren. In 22 der 117 Fehlerberichte, die von Dokumentationen handeln, wird angegeben, dass die Dokumentation eines oder mehrerer Elemente fehlen, diese also nicht beschrieben sind.
Grob jeder zehnte der analysierten Fehlerberichte dreht sich um die (Un-)Sichtbarkeit gewisser Elemente der API. In vielen Fällen fordern Nutzer der API Schnittstellen, über die sie auf interne Klassen oder Attribute zugreifen können.
8,4% der API-Usability betreffenden Fehlerberichte beschäftigen sich mit der Speicherverwaltung. Meistens geht es dabei um Speicherlecks, die durch die API verursacht werden.
In ungefähr gleich vielen Fällen (8,2%) geht es um die Anzahl und Art der Funktionsparameter oder um den Rückgabewert. Zu viele obligatorische Parameter, die unter Umständen auch schwer zu verstehende Parametertypen beinhalten, können den API-Nutzer irritieren, genauso wie unpassende oder fehlende Rückgabewerte.
42 Mal (also in etwa 7,5% der Fälle) wurden Fehler im Zusammenhang mit anderen Plattformen und APIs gemeldet. Für eine hohe Benutzerfreundlichkeit sollten APIs möglichst plattformübergreifend funktionieren, nicht von anderen APIs abhängen und mit anderen APIs nicht ungewollt interferieren. Sind solche Abhängigkeiten doch vorhanden, sollte das an mehreren Stellen verdeutlicht werden.
Es gab auch Fehlerberichte über Probleme mit neuen Versionen (circa 6%). Abwärtskompatibilität ist in APIs sehr wichtig, gerade wenn die API öfter Updates bekommt.

A.2 Ergebnisse der Studie von Grill et al. (2012)

Im Folgenden werden die Ergebnisse der Studie von Grill et al. zusammengefasst, die in Kapitel 3.2 vorgestellt wurde.
Während der *heuristischen Evaluation* sind insgesamt 127 einzigartige Usability-Probleme ermittelt worden. Der durchschnittliche Schwierigkeitsgrad bzw. das durchschnittliche Ausmaß der Probleme betrug 3,1, wobei 1 für harmlos und 5 für folgenschwer steht. Die Kategorie, unter der die meisten Probleme fielen, ist die Dokumentation. 35 Mal wurde eine fehlende, falsche oder unvollständige Stelle in der Dokumentation gefunden. Für die restlichen Ergebnisse, die in dieser Phase entdeckt wurden, siehe Anhang A.2 Die zweitmeisten Probleme (26) betrafen die Komplexität. Manche der Klassen sollten umstrukturiert oder sogar entfernt werden, um die Komplexität der API zu verringern. Platz drei teilen sich die Kategorien Benennung und Konsistenz mit je 13 Einträgen. Manche API-Elemente waren sehr eigenartig benannt, sodass sie den Nutzer verwirren. Außerdem gab es Inkonsistenzen, was Konventionen betrifft, wie das Fehlen von Gettern und/oder Settern an manchen Stellen.
Auch im *Workshop* gab es die meisten Usability-Probleme in der Dokumentation, nämlich 12 der insgesamt 44 im Workshop registrierten Usability-Probleme. Sieben Probleme wurden der konzeptionellen Korrektheit zugeordnet. Dass eine Klasse, von der ein Objekt erzeugt werden muss, um eine Funktion dieser Klasse zu benutzen, nach außen hin unsichtbar (private) ist, ist ein Beispiel für solch eine fehlende konzeptionelle Korrektheit. 6 der 44 Usability-Probleme kamen von Designentscheidungen, bei der die Nutzerperspektive nicht beachtet wurde. Wenn z.B. `createTV(True, int size)` eine Funktion ist, die einen Fernseher der Größe `size` erzeugt, und es sich um einen Farbfernseher handelt, weil der erste Parameter auf `True` gesetzt ist, kann es den Nutzer verwirren. `createTV(boolean colorTV, int size)` oder `createColorTV(int size)` wäre angebrachter.