FH MÜNSTER
University of Applied Sciences

Fachbereich
Energie · Gebäude · Umwelt

Bereitstellung von Open Source Software in der Energiewirtschaft – Ein Leitfaden

Bachelorarbeit

26. August 2022
Gregor Becker
gregor.becker@fh-muenster.de

Erstprüfer: Prof. Dr.-Ing. Peter Vennemann
Zweitprüfer: Dr. rer. nat. Jannik Hüls

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
BSD	Berkley Software Distribution
CDDL	Common Development and Distribution License
CI	Continuous Integration
CLI	Command Line Interface
CVCS	Centralized Version Control System
CVS	Concurrent Version System
DOI	Digital Object Identifier
DVCS	Distributed Version Control System
EPL	Eclipse Public License
ESM	Energiesystemmodellierung
FAIR	Findable, Accessible, Interoperable and Reusable
FAQ	Frequently Asked Questions
FSF	Free Software Foundation
FuE	Forschung und Entwicklung
GPL	General Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
JORS	Journal of Open Research Software
JOSS	The Journal of Open Source Software
LAN	Local Area Network
LGPL	Lesser General Public License
MPL	Mozilla Public License
oemof	open energy modelling framework
OS	Open Source
OSI	Open Source Initiative
OSS	Open Source Software
PEP	Python Enhancement Proposals
PR	Pull Request
QM	Qualitätsmanagement
RCS	Revision Control System
reST	ReStructuredText
RTD	Read the Docs
SCM	Source Code Managment

SESMG Spreadsheet Energy System Model Generator
SRS Software Requirements Specifications
SVN (Apache) Subversion
UI User Interface
VCS Version Control System

Zusammenfassung

Programmierung von Open Source Software in der Energiewirtschaft nimmt seit Beginn der 2000er stetig zu. Dies gilt sowohl für den Bereich der Forschung und Entwicklung, als auch für die Industrie und Wirtschaft. So werden beispielsweise Modelle zur Planung und Optimierung von Energiesystemen umgesetzt. Eine Open Source Veröffentlichung ist in diesem Forschungsfeld besonders wichtig, um die Überprüfbarkeit von Modellannahmen sowie der Vergleichbarkeit verschiedener Modellansätze zu garantieren. Einer Open-Source Veröffentlichung stehen jedoch häufig die Hürden von hohem Fristendrucks, fehlender Finanzierung und fehlendem Detailwissen der Publizierenden entgegen. Deshalb bleiben diese Softwareprodukte meist im Entwurfsstadium und sind daher schwierig wieder zu verwenden.

Mithilfe des neu erarbeiteten Schritt-für-Schritt Leitfadens zur standardisierten Implementierung einer Open Source Software, wird die Hürde und der zeitliche Aufwand zur Standardisierung von Open Source Repositories weitestgehend reduziert. Hierbei wird für jedem Bestandteil des zu standardisierenden Repositorys eine umfassende Erklärung der üblichen Standards sowie eine Empfehlung für unterstützende Softwarelösungen ausgesprochen.

Der Leitfaden orientiert sich an den aus der ISO 12207 resultierenden Phasen des Softwarelebenszyklus und ermöglicht einen Einstieg zu jedem Entwicklungsstand der Software. Seine grafische Aufbereitung in Form eines Prozessablaufplans erleichtert die Einschätzung des individuellen Status der Standardisierung eines vorliegenden Open Source Projektes. Als Treiber der Standardisierung eines Open Source Projektes sind insbesondere die bessere Lesbarkeit, Wartbarkeit und Testbarkeit der standardisierten Open Source Software wichtig.

Bei der Anwendung auf das bereits bestehende Open Source Projekt des Spreadsheet Energy System Model Genarators fiel auf, dass ein verspäteter Einstieg in ein systematisches Vorgehen (wie er mit dem Leitfaden dieser Arbeit gegeben wird) zu erheblichen Mehraufwand bei der Standardisierung führen kann. Dennoch konnten im Zuge der Umsetzung des erarbeiteten Leitfadens weitreichende Verbesserungen des Projektes vor dem Hintergrund der Standardisierung erreicht werden (z. B. Versionierung & Wartbarkeit).

Insgesamt lässt sich festhalten, dass eine frühestmögliche Standardisierung der Open Source Repositories durchgeführt werden sollte, um spätere Mehrarbeit zu vermeiden und die frühstmögliche Wiederverwendbarkeit für Dritte zu gewährleisten.

1 Einleitung

Das Open Source (OS)-Prinzip prägt in den vergangenen Jahren die Softwareentwicklung in der Wirtschaft sowie in der Forschung und Entwicklung (FuE). So nutzen 49 % aller Mitarbeitenden aus schweizer Unternehmen und Behörden Open Source Software (OSS) in mehr als der Hälfte ihrer täglichen Tätigkeiten [1]. Die steigende Bedeutung resultiert aus Vorteilen wie der Erhöhung der Langlebigkeit, der persönlicher Reputation und dem steigenden Bekanntheitsgrad des Softwareprodukts [2]. Im Bereich der FuE kann die steigende Bedeutung der OSS unter anderem auf das Findable, Accessible, Interoperable and Reusable (FAIR)-Prinzip [3] zurück geführt werden. In der Energiewirtschaft sind Open-Source-Veröffentlichungen besonders relevant, um (i) Ergebnisse miteinander vergleichen zu können, (ii) Stärken und Schwächen einzelner Methoden zu erkennen und (iii) um Reproduzier- und Validierbarkeit von Forschungsergebnissen zu gewährleisten [3–5].

Dem entgegen stehen die durch die OSS verursachten Herausforderungen, die die Publikation im Bereich der FuE erschweren. Zum Beispiel reicht das reine Softwareprodukt meist nicht aus, um den nicht-funktionalen Anforderungen der Stakeholder am Softwareprodukt (Weiterentwicklende, Nutzende und Publizierende) gerecht zu werden [6, 7]. Dies führt dazu, dass sowohl in der Energietechnik als auch in anderen Ingenieurwissenschaften, wo das Softwareprodukt meist nur als Werkzeug zur Erstellung von Forschungsergebnissen anzusehen ist, die Softwarelösungen und -packages aus Abschlussarbeiten oder Promotionsvorhaben im Draft-Status (Entwurfszustand [8]) verbleiben [2]. Eine umfassende Prüfung des selbst programmierten Quellcodes ist zeitaufwändig und bringt das eigentliche Forschungsvorhaben nicht voran. So wird bei der Kostenkalkulation von Forschungsvorhaben meist der Aufwand der Programmierung eingeplant und somit auch vergütet. Die Standardisierung und Weiterentwicklung des Softwareproduktes werden jedoch an die Community weitergegeben [2].

Aufgrund der Herausforderungen in der FuE, adressiert der aus dieser Arbeit resultierende Leitfaden Studierende und Mitarbeitende des naturwissenschaftlich-technischen Bereichs, die in einer Abschlussarbeit oder ihrer Tätigkeit ein Softwareprodukt programmieren. Darüber hinaus kann der Leitfaden als Nachschlagewerk bei der Teilhabe an einem OSS-Projekt verwendet werden.

Der Zeitaufwand des Standardisierungsprozesses, soll durch die vorliegende Arbeit reduziert werden, sodass die Hürde zur offenen Publikation des Softwareproduktes, die ein Problem innerhalb der FuE darstellt verringert wird [5]. Die vorliegende Thesis stellt einen Leitfaden (siehe Kapitel 3) zur Standardisierung des selbst geschriebenen Quellcodes dar.
Im Rahmen dieser Thesis werden die folgenden drei Forschungsfragen beantwortet.

1. Welche **Schritte** sind vom Draft-Status **zur standardisierten Software** nötig?

2. Wie können die **nicht-funktionalen Anforderungen** für unterschiedliche Interessensgruppen erfüllt werden und bestehen zwischen ihnen **Konflikte**?
3. Ist am Ende des Standardisierungsprozess ein Softwareprodukt **publikationsfähig**?

Auf Grundlage einer Literaturrecherche, die in Kapitel 2 zunächst beschrieben wird, werden notwendige Maßnahmen (z. B. Docstring-Standardisierung) sowie Grundsatzentscheidungen (z. B. Wahl des Version Control System (VCS) und der Lizenz) zur Überführung des Softwareprodukts aus dem Draft-Status zur standardisierten Software erarbeitet (siehe Kapitel 3). Anhand des Anwendungsbeispiel des Spreadsheet Energy System Model Generators (SESMGs) (siehe Kapitel 4) werden die vorher erarbeiteten Empfehlungen des Leitfadens geprüft.

Der in dieser Arbeit erstellte Leitfaden legt den Fokus auf die Python-Programmierung, da diese die am meisten genutzte dynamische Programmiersprache in der Energietechnik und -wirtschaft darstellt [9]. Dennoch sind weite Teile der Arbeit auch auf die Entwicklung mit anderen Programmiersprachen übertragbar.

2 Methode

Bevor der Leitfaden dargestellt und die Inhalten eines standardisierten VCS-Repositorys betrachtet werden, wird zu Beginn erst eine Definition des Begriffs Leitfaden im Sinne dieser Arbeit vollzogen und anschließend das methodische Vorgehen der Erstellung sowie der Prüfung beschrieben.

Definition Leitfaden:
Der Begriff Leitfaden wird in der Literatur unterschiedlich definiert. Für die vorliegende Arbeit wird sich auf folgende Definition bezogen [10]:

> *„[Ein Leitfaden ist eine] kurz gefasste Darstellung zur Einführung in ein Wissensgebiet.“*

Der in Kapitel 3 auffindbare Leitfaden resultiert wie bereits in der Einleitung erwähnt aus einer umfangreichen Literaturrecherche. Hierbei werden zu Beginn alle Bestandteile einer OSS identifiziert, in dem zum einen Leitfäden zur Programmierung bzw. zum Betrieb einer OSS gesichtet werden z. B. [11] und zudem ein Betrachtung einiger großer Repositorys – groß meint in diesem Zusammenhang eine große Community – durchgeführt wird.

Daraufhin wird zu den einzelnen Unterkapiteln des Leitfadens (siehe Kapitel 3) eine tiefergehende Literaturrecherche durchgeführt, sodass sich ein umfassendes Bild über den jeweiligen Standardisierungsschritt erreichen lässt.

Nachdem die Informationslage der jeweiligen Prozessschritte ausreichend ist, werden diese in einem dritten Schritt dem in ISO 12207 definierten Softwarelebenszyklus zugeordnet, sodass für die Publizierenden ersichtlich wird, zu welchem Zeitpunkt des Entwicklungsprozesses der beschriebene Bestandteil zu erstellen bzw. zu standardisieren ist. Der Softwarelebenszyklus setzt sich aus den folgenden sieben Phasen zusammen:

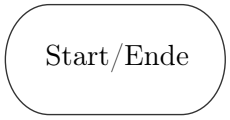
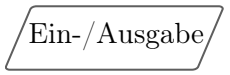
- Anforderungen & Spezifikation
- Planung
- Entwurf & Design
- Implementierung & Integration
- Betrieb & Wartung
- Stilllegung
- Qualitätsmanagement (QM)

Diese lassen sich jedoch basierend auf dem Best Practice Leitfaden der Softwareentwicklung von Schatten et al. in vier Oberkategorien zusammenfassen, wobei sich das Qualitätsmanagement (QM) über den gesamten Verlauf des Softwarelebenszyklus erstreckt. Die vier Oberkategorien, die die Basis des Ablaufes des Leitfaden (Spalte 2) darstellen sind

- Softwarespezifikation,
- Design & Implementierung,
- Softwarevalidierung, und
- Softwareevolution.

Nachdem das methodische Vorgehen der Recherche zur Erstellung des Leitfadens aufgegriffen wurde, seien nun noch die Form- und Farbgebung des Selbigen erläutert. Für die Formgebung wurde sich den Formen der DIN 66001 – Norm zur Erstellung von Programmablaufplänen – [12] bedient, welche in Tabelle 2.1 kurz beschrieben werden.

Tabelle 2.1: Formen der DIN 66001 und ihre Bedeutung [13].

	gerichtete Verbindung zweier Prozessschritte
	Start/Ende des Prozesses
	Anweisung oder Prozessschritt des Prozessablaufes
	Bedingte Fortführung des Prozesses (üblichste Form: logische Verzweigung, mit Ja/Nein Bedingungen)
	Eingabe der oder Ausgabe an die Nutzenden
	Aufruf eines Teilabschnittes des Prozesses (symbolisiert mehrere hintereinanderliegende Prozesse, die zu einem Teil- bzw. Unterprozess zusammengefasst werden)

Die im Leitfaden auffindbaren Farben verbildlichen die adressierte Interessengemeinschaft, sodass ersichtlich ist, wem die Änderung am Softwareprodukt, der Dokumentation oder den VCS-Einstellungen zu Gute kommt. Des Weiteren ermöglicht es eine Abwägung, ob diese Interessengemeinschaft von hoher oder geringer Wichtigkeit für das eigene Softwareprodukt ist. Je nach Bewertung kann der Prozessabschnitt intensiver oder weniger intensiv bearbeitet werden.

Anwendungsbeispiel SESMG:
Der SESMG [14] ist eine OSS, basierend auf dem open energy modelling framework (oemof). Sie ermöglicht Planenden, die Modellierung und Optimierung urbaner Energiesystemen (also Stadtquartieren, -vierteln oder Dörfern) ohne Kenntnisse der Programmierung anwenden zu müssen. Dies ist durch eine tabellenbasierte Schnittstelle zum Quellcode möglich [15]. Die Energiesystemmodellierung (ESM) zielt darauf ab, Minima vordefinierter Zielwerte zu identifizieren. Mithilfe multi-kriterieller Ansätze, wie der ϵ-Constraint-Methode (Pareto-Optimierung) ist es nicht nur möglich, das volkswirtschaft-

lich günstigste oder das emissionsärmste Szenario zu bestimmen, sondern auch pareto-optimale Szenarien (z. B. günstigstes Szenario bei 30 % CO_2-Reduktion) zu bestimmen [15]. Die erste Verison des SESMGs wurde im Kontext der Masterarbeit von Christian Klemm entwickelt, und seitdem im Drittmittelforschungsprojekt „R2Q – RessourcenPlan im Quartier“ [16] stetig weiterentwickelt.

3 Leitfaden

Abbildung 3.1 zeigt den aus der Literaturrecherche resultierenden Leitfaden zur Standardisierung einer OSS. Hierbei lässt sich zunächst festhalten, dass es sich wie in Kapitel 2 beschrieben um einen Softwarelebenszyklus handelt, folglich der Verlauf in mehreren Wiederholungen durchlaufen wird. Bevor genauer auf Aufbau und Funktion des Leitfadens (Abbildung 3.1) eingegangen wird, sei an dieser Stelle herausgestellt, dass in den folgenden Unterkapiteln die einzelnen Prozessschritte detailiert beschrieben bzw. erarbeitet werden.

Die Spalten 1 und 2 des dreispaltigen Layouts (siehe Abbildung 3.1) dienen hauptsächlich der Orientierung, sodass zum einen klar ist zu welchem Zweck der betrachtete Teil der Standardisierung verfolgt wird (Spalte 1) und zum anderen in welche Oberkategorie des in Kapitel 2 beschriebenen Softwarelebenszyklus die betrachtete Maßnahme fällt. Spalte 3 hingegen stellt den Leitfaden zur Standardisierung einer OSS dar.

Ein Einstieg in den vorliegenden Leitfaden ist an vier verschiendenen Punkten möglich. Diese vier Punkte beschreiben verschiedene Zustände der zu standardisierenden OSS, die im Folgenden definiert werden.

Einstiegspunkt 1: Stehen die Publizierenden am Beginn der Planung der Suche bzw. der Entwicklung einer OSS und sind folglich weder die Anforderungen definiert noch Quellcode implementiert, so sollte dieser Einstiegspunkt gewählt werden.

Einstiegspunkt 2: Dieser Einstiegspunkt ist zu wählen, wenn bei bereits existierender Definition der Anforderungen bei der Sichtung des Marktes herausgefunden wurde, dass es keine verwandte OSS gibt, die mit akzeptablem Aufwand die Anforderungen erfüllt.

Einstiegspunkt 3: Wurde anders als bei Einstiegspunkt 2 eine verwandte OSS gefunden, die mit akzeptablem Aufwand die festgelegten Anforderungen erfüllt, sollte dieser Einstiegspunkt gewählt und eine Kollaboration am VCS-Repository angestrebt werden.

Einstiegspunkt 4: Existiert bereits ein VCS-Repository, welches aufgrund mangelnder Kenntnis und Fristendruck nicht standardisiert wurde, sollte dieser Einstiegspunkt den Startpunkt im Leitfaden darstellen.

Konnte der Zustand der zu standardisierenden Software einem der Einstieg zugeordnet werden, so beginnt der Prozess am jeweiligen Startpunkt, wobei wie bereits erwähnt jeder Prozessschritt des Leitfadens, bei dem weitergehendes Wissen notwendig ist in einem Unterkapitel zu finden und per Klick auf die jeweilige Prozessschritt zu erreichen ist.

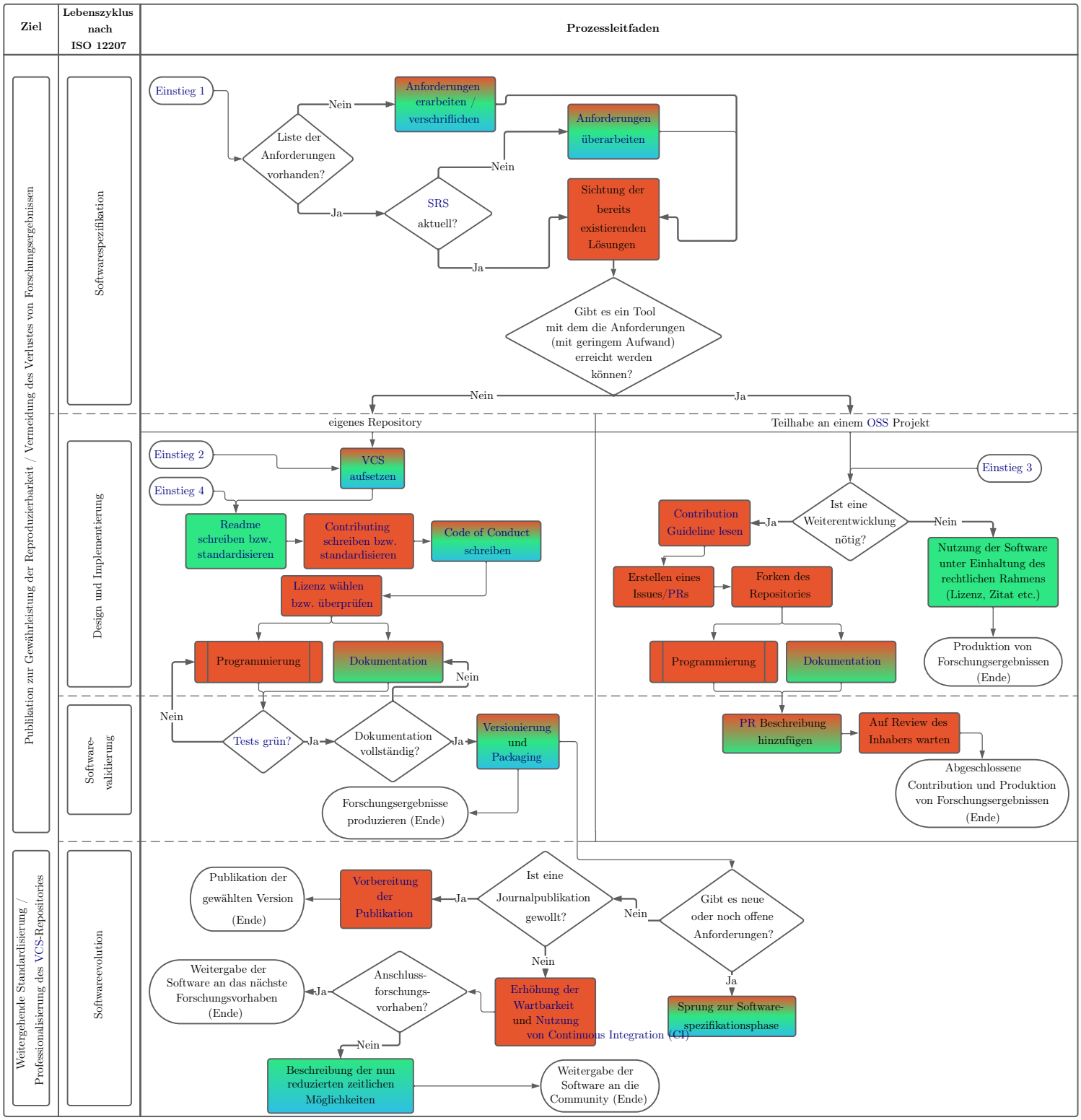


Abbildung 3.1: Prozessleitfaden (selbst erstellte Grafik).

3.1 Interessengemeinschaften

Zunächst werden die Interessengemeinschaften an einem Softwareprodukt betrachtet. Dies soll dazu dienen, im Folgenden Interessenskonflikte, die nach Axelsson und Skoglund [17] zwischen den Interessengruppen üblich sind, zu identifizieren.
Kilamo et al. [18] beschreiben die hierachische Struktur im Bereich der OSS-Entwicklung als „Zwiebelmodell“, welches neben der projektleitenden Person im Kern aus folgenden Interessengemeinschaften gennant:

- Kernmitglieder
- aktive Entwickelnde
- peripherer Entwickelnde
- Fehler-behebende Nutzende
- Fehler-meldende Nutzende
- Lesende
- passive Nutzende

Hierbei werden die Gruppen mit fallender Verantwortung immer größer, was das Sinnbild der Zwiebel (größer werden Schalen bei größerer Distanz vom Inneren) erklärt.

Die Interessengemeinschaften die im Zwiebelmodell sehr fein untergliedert werden, können in die Überkategorien

- Publizierende, und
- Community & Andere

geclustert werden. Anders ist es bei der Überkategorie Industrie & Wirtschaft. Diese fasst nicht Interessengemeinschaften des Zwiebelmodells zusammen, sondern beeinflusst das Softwarepaket sowie sein Ökosystem von außerhalb [18].

Definition Software Ökosystem:

> *Das Ökosystem einer Software beschreibt das „synergetische [...] Miteinander von Organismen, die von der Existenz und dem Handeln der Anderen profitieren“ [19]. Dies basiert im zentralen auf einer technischen Plattform wie zum Beispiel einem Betriebssystem oder einem VCS.*

Unter die **Publizierenden** fallen alle die, die aktiv zum Quellcode, zum Repository (siehe Kapitel 3.3) oder zur Dokumentation beitragen. Hierunter verstehen Nakakoji et al. [20] den Kern der Entwicklung, worunter zum einen die Initiierenden und zu Beginn der Entwicklung meist etwa drei bis sieben Entwickelnde zählen [21].

Die zweiten Gruppe (**Community und Andere**), der die unteren hierachischen Gruppen des „Zwiebelmodells“ zuzuordnen sind, begünstigt den Entwicklungsprozess der OSS. Sie liefern Rückmeldungen und beheben kleinere Fehler, was einen der Kernaspekte und Hauptvorteile von OS-Entwicklung darstellt [17].

Die dritte und letzte Gruppe stellt die **Industrie und Wirtschaft** dar, welche aufgrund professioneller Entwicklungsteams, die neben der Entwicklung von OSS auch an proprietärer Software arbeiten, Erfahrung und Know-How mitbringt. Sie kann Einfluss auf den inneren Kreis der Publizierenden nehmen, was einerseits möglich ist, in dem Publizierende beim jeweiligen Unternehmen angestellt sind, oder diese andererseits durch Ratschläge unterstützen [18]. Unabhängig von der Art und Weise, wie auf die Publizierenden Einfluss genommen wird, können bei Industrie- und Wirtschaftspartnern zwei verschiedene Typen unterschieden werden. Zum einen gibt es die enthusiastischen Partner, die den Mehrwert der OSS für ihr Unternehmen wahrnehmen und deshalb gewillt sind die OSS voran zu bringen. Und zum anderen die pessimistischen Partner, welche aufgrund der Änderung in ihren Arbeitsabläufen zögern ein Softwarerprodukt OS zu veröffentlichen [18]. Anhand dieser drei übergeordneten Interessengruppen, werden im Weiteren die (nicht-)funktionalen Anforderungen an das Softwareprodukt erarbeitet. Diese werden, wie bereits in Kapitel 2 beschrieben mit drei verschiedenen Farben (siehe Abbildung 3.1) im Leitfaden dargestellt.

3.1.1 Kommunikation zwischen Interessengemeinschaften

Die OS-Entwicklung lebt von der Kommunikation zwischen den Interessengemeinschaften. Dies liegt daran, dass zum einen gefundene Fehler oder eine Anfrage für eine Erweiterung der Funktion an die Publizierenden herangetragen werden müssen. Jedoch auch umgekehrt eine Beschreibung der Änderungen an der OSS transparent an die Nutzenden übermittelt werden muss. Denkbare Formen des Kommunikationsweges sind Github Diskussionen, Mattermost, Element oder ähnliche Kommunikationsserver. Hierbei sollte jedoch Wert auf einen geregelten Umgang gelegt werden, dessen Grundsatz der Code of Conduct darstellt (siehe Kapitel 3.4.3).

3.2 Softwarespezifikationsphase

Die Softwarespezifikationsphase ist der Beginn des Softwarelebenszyklus (siehe Kapitel 2) [22]. Darin werden zuerst die technischen/ingenieurwissenschaftlichen Anforderungen an das neu zu schaffende oder zu suchende Softwareprodukt erarbeitet. Ein geeignetes Werkzeug für die Erarbeitung der Anforderungen an die OSS sind die Software Requirements Specifications (SRS) [23]. Hierbei wird durch die schriftliche Beantwortung der folgenden Fragen, die SRS und somit die Anforderungen an die OSS erarbeitet.

- **Funktionsweise**: Was soll das Softwareprodukt leisten?
- **Interfaces/Integration**: Wie interagiert das Softwareprodukt mit Menschen, mit der Hardware des Systems, mit anderer Hardware und anderer Software bzw. wie lässt sich das Softwareprodukt in bereits bestehende Produkte integrieren?
- **Performance**: Welche Anforderungen gibt es an die Geschwindigkeit, die Verfügbarkeit, die Reaktionszeit, die Wiederherstellungszeit der verschiedenen Softwarefunktionen usw.?
- **Eigenschaften**: Welche Anforderungen gibt es an Portabilität, Korrektheit (Genauigkeit & Fehlerfreiheit), Wartbarkeit, Sicherheit, usw.?
- **Design-Einschränkungen**: Gelten bestimmte Standards, die Implementierungssprache, Richtlinien für die Datenbankintegrität, Ressourcenbeschränkungen, Betriebsumgebung(en) usw.?

Darüber hinaus dient die SRS besonders dem QM, da im Verlaufe des Softwarelebenzyklus (siehe Kapitel 2), die Anforderungen der OSS stets zu überprüfen sind [23]. Demnach kennzeichnet sich eine gute SRS durch Erfüllung der folgenden Merkmale:

- Richtigkeit
- Unzweideutigkeit (keinen Auslegungsspielraum einer Anforderung)
- Vollständigkeit
- Einheitlichkeit/Stringenz
- Relevanz
- Nachvollziehbarkeit
- Modifizierbarkeit
- Rückverfolgbarkeit

Im Anschluss an die Erstellung der SRS folgt die Sichtung bereits existierender Softwareprodukte. Hierbei wird überprüft, ob bereits eine OSS-Lösung existiert, die mit keinem oder geringem Programmieraufwand in der Lage ist die Anforderungen zu erfüllen [24, 25]. Ist dies nicht der Fall, sollte möglichst früh ein eigenes Repository in einem gewählten VCS veröffentlicht werden (siehe Kapitel 3.3). Dadurch wird für andere Entwickelnde klar, dass in diesem Themenbereich gearbeitet wird. Dies beugt dem Risiko doppelter Programmierung vor [24]. Sollte bei der Sichtung bestehender Softwareprodukte ein geeignetes gefunden worden sein, so ist eine Kollaboration am bestehenden Produkt anzustreben.

3.3 Version Control System

Resultiert aus der Softwarespezifikationsphase (siehe Kapitel 3.2), dass eine neue Software zu programmieren werden soll, muss ein geeignetes Version Control System (VCS) gewählt werden.

Definition

> *Ein VCS ist nach Zolkifli et al. [26] ein System, das die Änderungen an Software, Dokumentation und anderen projektrelevanten Dateien festhält, und somit ermöglicht, Informationen wie Änderungszeitpunkte und -autor:innen nachzuvollziehen [27].*

Hierdurch werden Prozesse wie die *Rückkehr zur letztfunktionierenden Version* oder die *Frage nach Hilfe bei den Letztautor:innen* möglich und/oder vereinfacht.
Neben dem Begriff VCS werden in der Literatur verschiedene Synonyme genutzt, die sich lediglich marginal in ihrem Fokus unterscheiden. Als Synonyme für VCS, werden in diesem Kontext

- Revision Control System (RCS),
- Software Configuration Managment,
- Source Code Managment (SCM) und
- Source Code Control

angeführt.

3.3.1 Typen

VCSs lassen sich nach Otte [28] in drei verschiedene Typen unterscheiden:

- Centralized Version Control Systems (CVCSs)
- rein lokale Versionsverwaltungen
- Distributed Version Control Systems (DVCSs)

Bei CVCS betriebener Verwaltung, findet die Versionsverwaltung ausschließlich auf einem zentralen Server statt, zu dem die Nutzenden eine stetige Verbindung (meist über das standortbetriebene Local Area Network (LAN)) benötigen. Des Weiteren befindet sich bei dieser Variante der Versionsverwaltung lediglich eine Version der zu verändernden Datei auf dem Rechner des Nutzenden, das einzige vollständige Repository liegt auf dem Server. Dieser Umstand und die Notwendigkeit der Netzwerkbindung können als Hürden zur überregionalen Zusammenarbeit (einem Ziel des FAIR-Prinzips als auch einer der wichtigen Bestandteil der OS-Entwicklung [26, 29]) angesehen werden. Gegensätzlich zum CVCS verbleiben bei der rein lokalen Versionsverwaltung die verschiedenen Versionen einer Datei auf der Festplatte der Entwicklenden [26, 29]. Dies kann zu einem Übersichts- und/oder Fortschrittsverlust (z. B. durch das Löschen einer falschen Datei) führen, sodass auch diese Art der Verisonsverwaltung nicht dem FAIR-Prinzip genügt [29]. DVCS-Plattformen stellen eine Hybridform der CVCS-Systeme und der rein lokalen Versionsverwaltung dar [26]. Diese Form des VCS ermöglicht das Offline-Arbeiten (Publizierende haben eine lokale Kopie des Repositorys) sowie das Arbeiten von einem beliebigen Standort (Zugriff über das Internet) und vereint somit die Stärken der beiden anderen Ansätze. Beispielplattformen für CVCS und DVCS sind in Tabelle 3.1 aufgeführt.

Tabelle 3.1: Beispiele für CVCS und DVCS Platformen

centralized version control system (CVCS)	distributed version control system (DVCS)
Concurrent Version System (CVS) [28, 29]	Git / Github [26, 28, 30]
(Apache) Subversion (SVN) [26, 28, 29]	Mercurial [26, 29]
Preforce [26]	Bazaar [26]
	BitKeeper [26]
	Darcs [26]
	Sourceforge [24]

Die Entwicklung der Relevanz CVCS und DVCS Platformen, lässt sich aus den Aufrufstatistik des Suchanbieters Google für die weltweite Suche von CVS, SVN und Git ableiten (siehe Abbildung 3.2) [31]. Was darauf beruht, das das Suchinteresse zu einem Thema in der Regel einen direkten Zusammenhang zur Relevanz hat. Auffällig ist, dass während die Bedeutung von SVN vor 2010 noch zunahm und das Interesse von CVS schnell übetraf, das Suchinteresse nach Git seit Mitte 2010 das Interesse an den CVCS dominiert. Für die Betrachtung der Abbildung 3.2 ist hinzuzufügen, dass der Einbruch im letzten Quartal 2020 und den nachfolgenden Monaten mit hoher Wahrscheinlichkeit auf die pandemiebedingten Einschränkungen zurückzuführen ist.

Die Entwicklung der DVCS (wie Git) lässt sich auf die oben beschriebenen Vorteile zurückführen, welche besonders im Bereich der OS-Entwicklung zum Tragen kommen. Daher wird die OS-Entwicklung als Treiber des Wandels von den CVCS zu den DVCS Platformen herausgestellt [26].

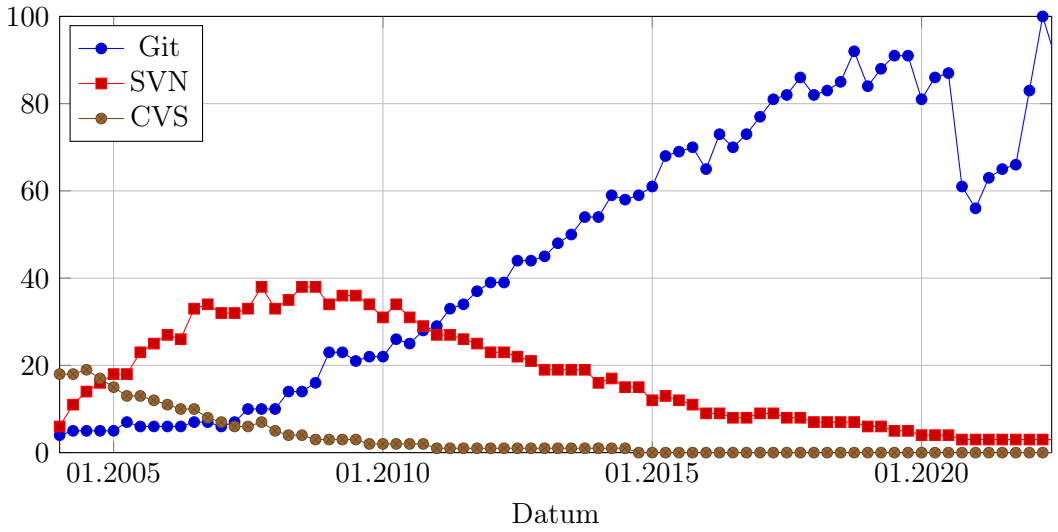


Abbildung 3.2: Entwicklung der VCS Suchanfragen (Google) [31].
Bezugswert: 100 = Zeitintervall mit dem höchsten Suchinteresse.

Basierend auf dieser Entwicklung, wird die Verwendung von Gitlab oder Github empfohlen. Aufgrund des Institutionsunabhängigkeit und der Nutzung im Anwendungsbeispiel werden sich die weiteren Betrachtungen ausschließlich auf Github beziehen.

3.3.2 Software zur Erstellung des Repositorys

Der hohe Funktionsumfang von Github und die Nutzung von Continuous Integration (CI) (siehe Kapitel 3.6) stellen OS-Enwickler vor eine hohe Herausforderung. Auch wenn die Erstellung des VCS-Repositorys auf den ersten Blick einfach erscheint, sollte bereits zu diesem Zeitpunkt eine standardisierte Repositorystruktur (siehe Kapitel 3.4 aufgesetzt werden, da diese sich nachträglich nur noch schwer ändern lässt. Dies steht der Empfehlung von Perez-Riverol et al. gegenüber, wonach ein Softwareprodukt so früh wie möglich in ein VCS zu überführen ist. Es ist damit zu begründen, dass die Publizierenden einer neu zu programmierenden Software sich zu Beginn mit der standardisierten Struktur eines Repositorys auseinander setzen müssen [30].

Abhilfe schaffen Repository Templates für verschiedene Programmierumgebungen der Platform Cookiecutter[1]. Zur Nutzung der in Tabelle 3.2 gegebenen Templates, werden lediglich *cookiecutter*, *tox* und *setuptools*, die sich über den Pythonpaketmanager *PyPI* installieren lassen, benötigt. Tabelle 3.2 zeigt die 5 relevanten auf Cookiecutter[1] zur Verfügung stehenden Pythontemplates und ihren Funktionsumfang. Für die Erstellung eines üblichen Python Packages wird an dieser Stelle die *Cookiecutter Pylibrary* aufgrund ihres hohen Funktionsumfangs empfohlen. Für Anwendungen, die eine strikt vom Betriebsystem getrennte Umgebung benötigen, sollte hingegen das Template *Python Package*

[1] https://www.cookiecutter.io

Template genutzt werden, da diese eine Dockervorbereitung beinhaltet. Neben den aufgelisteten Repositorytemplates sind im Laufe der Zeit noch einige durch die Community entwickelte Templates hinzugekommen, die jedoch im Rahmemn dieser Arbeit nicht weiter thematisiert werden sollen.

Tabelle 3.2: Cookiecutter Python Repository Templates bereitgestellt auf https://www.cookiecutter.io.

Template	Pythonversion	Nur für Github?	Versionsmanagement	Lizenzwahl	Pip-Vorbereitung	Docker-Vorbereitung	Linter	Pre-commit formatter	CLI	CI	Sphinx-Dokumentation	mkdocs-Dokumentation	Github Actions	requires.io
Cookiecutter Py-library	>=3.6	✗	✓	✓	✓	✗	flake8, pylama	black, blue, no	plain, argparse, click, no	Coveralls, Codecov, Scrutinizer, Codacy, Codeclimate, Travis, Appveyor	✓	✗	✓	✓
Cookiecutter Pypackage	>=3.6	✓	✗	✓	✓	✗	–	black	click, argparse	Travis, Tox	✓	✗	✗	✗
Cookiecutter Py-test Plugin	>=3.5	✓	✗	✓	✓	✗	–	–	–	Appveyor, Tox	✓	✓	✗	✗
Python Package Template	>=3.7	✓	✓	✓	✓	✓	mypy, darglint	pyupgrade, isort, black	–	Dependabot, Build Test, Greeting, release-drafter	✗	✗	✓	✗
Template Python	>=3.8	✓	✗	✗	✗	✗	pylint, pydocstyle	isort, black	–	Coverage	✗	✓	✓	✗

3.4 Standardisierte Repositorystruktur

Ein Repository stellt die Verzeichnisstruktur eines Projekts auf den Servern des VCS dar, auf welche in der Regel mehrere Publizierende (Collaborator) bearbeitenden Zugriff haben. Im Folgenden wird die Ordnerstruktur in Abbildung 3.3 als standardisierte Verzeichnisstruktur aufgegriffen.

Abbildung 3.3: Standardisierte Verzeichnisstruktur eines Repositories.

Im Hauptverzeichnis sind die *README.md*, die *CONTRIBUTING.md* und die *CODE_OF_CONDUCT.md* vorzufinden, diese Stellen zum einen die Startseite des VCS-Repositorys dar. Und dienen zum anderen der Beschreibung der Nutzung sowie des Mitwirkens am Repository.

3.4.1 Readme

Die *README.md* liegt gemäß der vorgegebenen Repositorystruktur (siehe Kapitel 3.4) im Hauptverzeichnis [32]. Sie ist notwendig, um das FAIR-Prinzip [3] zu erfüllen, da sie den Einstieg in das Softwareprodukt erleichtert (FAIR-Prinzip: Reusable). Darüber hinaus werden weiterführenden Informationen, wie z.B. die Dokumentation auffindbar angegeben (FAIR-Prinzip: Findable) [33]. Prana et al. halten fest, dass in der *README.md* die folgenden sieben Kategorien zu beschreiben sind.

- Der Abschnitt **„Was?“** stellt eine Kurzbeschreibung des vorliegenden Projektes dar. Diese sollte die in der Softwarespezifikationsphase erarbeiteten Informationen der SRS beinhalten, da diese Anforderungen und Ziele des vorliegenden Repositorys darstellt (siehe Kapitel 3.2) [22]. Somit ist ein schneller Überblick über Ziel und Funktionalität des Repositorys möglich, was die Grundlage zur Entscheidung, ob eine Kollaboration oder eine Auseinandersetzung mit dem Repository sinnvoll ist [27], darstellt. Des Weiteren sollte eine Einordnung des Softwareproduktes in den wissenschaftlichen Gesamtkontext vorgenommen werden.

- Daraufhin sollte der Nutzen des vorliegenden Softwareproduktes im Abschnitt **„Warum?“** beschrieben und seine Vorteile bzw. die Weiterentwicklung im Vergleich zu konkurierenden Softwareprodukten herausgestellt werden [32].
- Der README-Bestandteil **„Wie?“** beschreibt die Erstinbetriebnahme des vorliegenden Softwareproduktes und stellt den wichtigsten Teil der *README.md* dar [27, 32]. Hierin werden die Installation (siehe Kapitel 3.10), die Systemanforderungen sowie mögliche bekannte Probleme während der Installation beschrieben. Zudem wird eine Liste der Dependencies bereitgestellt. Dies soll die Erstinstallation sowie die weiterführende Nutzung vereinfachen [34]. Zuletzt soll eine Beschreibung des Ziels der im Repository enthaltenen Beispiele vorliegen, sodass den Nutzenden der Einstieg in die Arbeit mit der OSS weitestgehend vereinfacht wird [32, 34]
- Der Abschnitt **„Wann?“** gilt der Transparenz sowie dem Qualitätsmanagement. Er beinhaltet eine Beschreibung des aktuellen Projektstatus, eine Beschreibung des letzten Releases sowie der noch bevorstehenden Entwicklung. Dies ermöglicht den Publizierenden, Kenntnis über den aktuellen Projektstand zu erhalten [27, 32].
- Eine Vorstellung des und eine Beschreibung der Kontaktaufnahme mit den Publizierenden ist im Abschnitt **„Wer?“** zu finden (siehe Kapitel 3.1.1). Ferner werden projektspezifische Aspekte wie z. B. das bezahlende Unternehmen/Ministerium (Acknowledgement), die gewählte Lizenz (siehe Kapitel 3.5) oder dem Code of Conduct (siehe Kapitel 3.4.3) transparent dargelegt [32, 34].
- Der Abschnitt **„Referenzen?“** enthält meist verschiedene Links, wie z. B. zu einer Supportplattform, zur Volltextdokumentation, zur Application Programming Interface (API)-Dokumentation sowie zu möglichen unter Verwendung der Software entstandenen Publikationen [27].
- Die *README.md* wird meist durch die *CONTRIBUTING.md* (siehe Kapitel 3.4.2) ergänzt, in der beschrieben wird, wie das Kollaborieren an der Quellcodeentwicklung, an der Dokumentation oder anderen Repositorybestandteilen möglich ist. In der Kategorie **Mitwirken?** wird diese lediglich referenziert [35].

Prana et al. bringen zwei Beispielrepositories (ParallelGit[2] & Sandstorm) an, die als Orientierung bei der Erarbeitung einer *README.md* zu empfehlen sind. Des Weiteren wird die Verwendung von Kategorie-Badges [36] empfohlen, da diese die Lesbarkeit sowie die Übersichtlichkeit der *README.md* verbessern.

3.4.2 Contributing

In der Markdowndatei *CONTRIBUTING.md* des Hauptverzeichnisses des VCS-Repositorys wird erläutert, wie externe Personen am Projekt kollaborieren können [37]. Darin sollte besonders auf einen freundlichen Ton geachtet werden, da dieser von einem besonderen Interesse an den Kollaborierenden zeugt [38]. Eine *CONTRIBUTING.md* sollte die folgenden fünf Aspekte, die Bestandteile des Contributionprozesses sind, abdecken [39]:

[2] https://github.com/freelunchcap/ParallelGit

- Die **Projektausrichtung** stellt den Einstieg für neue Kollaborierende dar und sollte daher kurz beschreiben, wie ein Issue eröffnet werden kann, wenn ein Problem auftritt [39]. Auch ist es sinnvoll darzustellen, wie eine Entwicklungsumgebung aufgesetzt werden kann, sodass eine Weiterentwicklung der OSS ermöglicht wird.

- Unter dem **Contribution Workflow** sollte eine Beschreibung des Prozesses, von der Idee der möglichen Weiterentwicklung/Ergänzung der OSS hin zum fertiggestellten Pull Request (PR) vorzufinden sein. Dies dient dazu, die Hürde zur Kollaboration am OS-Repository weiter zu reduzieren [39]. Hierunter fallen z. B. eine Beschreibung des Aufbaus eines Issues oder eines PRs. Auch ist eine Aussage über die Notwendigkeit einer Contributor License Agreement (Bedingungen an den Beitrag geistigen Eigentums in das Repository [40]) notwendig.
  Im Rahmen des **Contribution Workflows** wird die Verwendung von Issue Templates sowie PR Templates empfohlen, da sie die Hürde zur Kollaboration reduzieren [30]. Issue Templates können durch einen automatisierten Github Prozess erstellt werden und dann den eigenen Ansprüchen angepasst werden [41]. Die Erstellung eines PR Templates hingegen muss manuell durchgeführt werden [42]. Hierzu helfen eine Vielzahl bereitgestellter Beispielen[3].

- Der Abschnitt **PR Kriterien** der *CONTIRBUTING.md* stellt eine Liste der von den Publizierenden beschlossenen Anforderungen an einen mergebaren PR dar.

- In diesem Abschnitt wird die Arbeitsweise der im vorliegenden Repository genutzten **CI-Tools** transparent beschrieben. Dadurch wird zusätzliche Erklärungen die zumeist sehr arbeits- und zeitintensiv sind, vorgebeugt [37, 39].

- Den letzten Abschnitt der *CONTIRBUTING.md* stellt die **Rückverfolgbarkeit/Traceability** dar. Dieser Abschnitt ermöglicht es die Entwicklung der Anforderungen sowohl vorwärts als auch rückwärts entlang des Softwarelebenszyklus zu verfolgen [39, 43].

Aufgrund der hohen Relevanz der Kollaboration externer Personen am VCS-Repository, wird empholfen, den Contributionprozess neben der *CONTRIBUTING.md* auch in der Dokumentation aufzugreifen (siehe Kapitel 3.9).

3.4.3 Code of Conduct

Die *CODE_OF_CONDUCT.md* hält die Ansprüche an das Verhalten aller Interessengemeinschaften (siehe Kapitel 3.1) am Softwareprodukt fest [44]. Hierin kann klargestellt, dass sich von Sexismus, Rassismus und Ähnlichem klar distanziert wird [2]. Zusätzlich wird transparent dargelegt, dass etwaige Aussagen z. B. im Diskussionsbereich des Repositorys mit Folgen wie Aufforderung zur Entschuldigung, zeitweiligen oder vollständigen Ausschluss aus dem VCS-Repository geahndet werden.

Nun kann zum Beispiel beim Eröffenen eines Issues eine Abfrage über die Kenntnisnahme der *CODE_OF_CONDUCT.md* hinzugefügt werden (siehe Kapitel 3.4.2). Dies ermöglicht, dass die Issue schreibende Person zeitgleich über das vorhanden sein eines *CODE_OF_CONDUCT.md* informiert wird.

[3] https://github.com/stevemao/github-issue-templates

Die Erstellung eines *CODE_OF_CONDUCT.md*, kann in Github durch das Hinzufügen einer neuen Datei mit dem Namen *CODE_OF_CONDUCT.md* mithilfe von auf Github bereitgestellten Vorlagen durchgeführt werden [45]. Nach Ersterstellung soll die Datei fortlaufend aktualisiert werden, sodass sie den Anforderungen und Anmerkungen der Nutzenden nach Austausch und Diskussion gerrecht wird. Dies sorgt dafür, dass der *CODE_OF_CONDUCT.md* unterstützend und nicht hindernd für den Prozess und das Arbeiten mit dem VCS-Repository ist [44].

3.5 Open Source-Lizensierung

Bereits mit der ersten Veröffentlichung eines Repository sollte eine Bereitstellungslizenz gewählt werden, da sie die Wiederverwendbarkeit der eigenen Software ermöglicht und gleichzeitig an Bedingungen knüpft.

Die Wahl der OS-Lizenz (voraus gesetzt es wurde sich vorher für eine OS-Publikation entschieden) des eigenen Softwareprodukts wird vornehmlich durch die Lizenzen der genutzten Dependencies beeinflusst. Daher ist es nötig, die üblichen Lizenztypen zu kennen. Kapitel 3.5.1 vergleicht diese hinsichtlich ihrer Eigenschaften sowie ihrer Kompatibilität. Abschließend wird ein Softwarepaket zur Überprüfung von Lizenzdiskrepanzen und zur der Wahl der eigenen Lizenz vorgestellt (siehe Kapitel 3.5.2).

3.5.1 Lizenztypen

Das Copyleft einer OS-Lizenz ist das Stellglied, mit dem die Anforderungen an bzw. die Erlaubnis für die Nutzung einer OSS und seinem Ökosystem (siehe Kapitel 3.1) beeinflusst werden [2, 18].

Definition Copyleft

> *Gegensätzlich zum Copyright stell das Copyleft Bedingungen an die Änderung bzw. Wiederveröffentlichung von OSS. Das Copyleft ermöglicht den Programmierenden die Modifikation des bereits Programmierten, solange die modifizierte Version unter den gleichen bzw. kompatiblen Rechten (Lizenzen) veröffentlicht wird [18, 46].*

Grundsätzlich kategorisiert die Literatur die OS-Lizenzen in zwei verschiedene Typen. Zum einen die akademischen und zum anderen die reziproken Lizenztypen, wobei bei reziproker Software die Stärke des hierdurch implizierten Copylefts, also die durch die Lizenz gestellten Anforderungen an die Wiederverwendung, variieren kann [47]. Die Freizügigkeit einer Lizenz teilt sich laut der Free Software Foundation (FSF) in die folgenden vier Freiheitsgrade auf [48]:

- 0. Freiheitsgrad: Software ausführbar für jeden Zweck
- 1. Freiheitsgrad: Anpassbarkeit von Quellcode auf eigene Problemstellung
- 2. Freiheitsgrad: Wiederveröffentlichung des veränderten Quellcodes
- 3. Freiheitsgrad: Erlaubnis, das Programm zu verbessern und die verbesserte Version wieder zu veröffentlichen

Während bei akademisch lizensierten Softwareprodukten alle Freiheitsgrade erfüllt werden, ist es bei reziproken abhängig von der Lizenzwahl. Die Freizügigkeit eines Softwareprodukts hat nur begrenzten Einfluss auf seine Kommerzialisierbarkeit, also die Möglichkeit es zu verkaufen. Während bei Lizenztypen wie Eclipse Public License (EPL) trotz eingeschränkter Freizügigkeit keine Bedingungen an den Verkauf des Softwareprodukts gestellt werden, legt die GNU General Public License (GPL) 3.0 einen maximalen Verkaufswert fest [49, 50]. Der letzte Aspekt, der Gegenüberstellung der meist gewählten Lizenztypen ist die Lizenzbindung (siehe Tabelle 3.3). Hierbei wird lediglich festgelegt, ob das Softwareprodukt in ein Paket mit einer anderen Lizenz eingebunden werden darf oder nicht. Meist gilt, dass das Copyleft des Softwareprodukts durch die Lizenzwahl des Gesamtproduktes nicht reduziert werden darf [18, 50]. Die FSF empfiehlt die Wahl des strengsten Copylefts, folglich die höchsten Anforderungen an die Wiederverwendung der OSS, da damit die Existenz der OSS am Stärksten geschützt ist. Den Verzicht auf Copyleft sollte laut der FSF ausschließlich bei Projekten mit weniger als 300 Zeilen in Anspruch genommen werden [50, 51]. In Tabelle 3.3 werden die Eigenschaften der aktuellen meist genutzten Lizenztypen resultierend aus dem „Report of License Proliferation“ der Open Source Initiative (OSI) [52] gegenüber gestellt.

Tabelle 3.3: Eigenschaften von Lizenztypen basierend auf [52].

Eigenschaft / Lizenztyp	Freizügigkeit (Copyleft)	Kommerzialisierbarkeit	Inkompatibilitäten	Lizenzbindung	Lizenzziel
	✓ : kein **(✓): schwach** **✗ : stark**	**✓: möglich** **(✓): möglich** **(mit Preisgrenze)** **✗: nicht möglich**		**✓: Ja** **✗: Nein**	
Apache 2.0	✓ [47, 49, 50, 53]	✓ [54]	GPL 2.0 [50]	✗ [54]	-
BSD 2-Clause (Free-BSD) 3-Clause (Modified-BSD)	✓ [50] ✓ [47, 49, 53]	✓ [50] ✓ [50]	- -	✗ [50] ✗ [50]	- -
CDDL	(✓) [50]	✓ [50]	GPL [50]	✗ [50]	-
EPL 2.0	(✓) [49, 50]	✓ [50]	GPL [49, 50]	✓ [49]	-
GNU GPL 3.0 LGPL	✗ [47, 53, 55] (✓) [47, 49, 53]	(✓) [49, 50, 55] ✓ [51]	OSS mit geringerem Copyleft GPL [50]	✓ [47, 56] ✓ [51]	freie Software [51] kommerzielle Software [51]
MIT (auch X11)	✓ [47, 53]	✓ [50]	-	✗ [50]	kleinere Anwendungen [50]
MPL 2.0	(✓) [47, 49, 53]	✓ [47]	GPL, LGPL [47, 49]	✓ [49]	-

Neben den in Tabelle 3.3 aufgeführten Softwarelizenzen stellen die FSF und die OSI auch Informationen über weniger etablierte und zweckgebundenen Lizenztypen zur Verfügung. Hierunter fallen z. B. Dokumentationslizenzen. Daher empfehlen sich in diesem Kapitel die Webseiten der FSF [50] und der OSI [57] als Nachschlagemöglichkeit.

3.5.2 Lizenzüberprüfungssoftware

Diese Vielfalt an Lizenzen führt häufig zu Lizenzdiskrepanzen, von denen die Publizierenden nicht umfassende Kenntnis haben. Dieses Problem löst ein Tool zur softwaregestützten Überprüfung der Lizenzkompatibilität. Eine solche Überprüfung ließt die *requirements.txt* (siehe Kapitel 3.3) aus. Nachdem die Lizenzinformationen heruntergeladen wurden, überprüft der Algorithmus unter Angabe der Wunschlizenz, ob es zu Inkompatibilitäten kommt. Die *requirements.txt* umfasst eine Auflistung der genutzten Dependencies, die um ihre Mindest- und oder Maximalversion ergänzt werden. Eine Beispielsoftware zur Lizenzüberprüfung ist das über den Pythonpaketmanager *PyPI* installierbaren Paket *LicenseCheck*[4] dar. Wird bei der Überprüfung eine Diskrepanzen gefunden, so müssen entweder die nicht zur gewünschten Lizenz passenden Dependencies ersetzt werden, oder es muss für das vorliegende OS-Projekt eine anderen Lizenz gewählt werden.

3.6 Continuous Integration

Definition

> *Continuous Integration (CI) resultiert ursprünglich aus dem Konzept des „extreme programming“ bei dem viele oder jeder der Publizierenden täglich eine Teilhabe am Repository anstrebt. Bei dieser Art des Programmierens, können Versionierung, Packaging usw. nicht mehr manuell umgesetzt werden und sollen daher von Workflows – kleine Automatisierungen, die über Einstiegssequenzen (z. B. Nutzende haben etwas gepusht) die Änderungen abgreifen – durchgeführt werden [58].*

Diese Workflows sammelt Github im Bereich der Github Actions [59], sodass bereits programmierte Workflows von anderen Nutzenden wiederverwendet und nicht neu erstellt werden müssen (FAIR-Prinzip). Typische Anwendungsfälle sind Neuberechnung der Testcoverage (siehe Software zur Berechnung und Erhöhung der Codecoverage), Überprüfung des Codestyle (siehe Software zur automatisierten Code Style Prüfung/Anpassung) oder auch Prozesse wie die automatisierte Versionierung (siehe Versionierungssoftware) sowie das Packaging für den Paketmanagmentplattformen wie *PyPI*. Die CI stellt also eine Automatisierung der stetig wiederkehrenden und fehleranfälligen Arbeitsschritte an einem Repository dar [58].

3.7 Versionierung

Wie bereits in Kapitel 3.3 herausgestellt wurde werden Versionskontrollsysteme dazu verwendet, die Entwicklung der Dateien des Repositories nachvollziehbar, rückverfolgbar und umkehrbar zu machen. Im Rahmen der Python Versionierung stellt der Python

[4] https://github.com/FHPythonUtils/LicenseCheck

Enhancement Proposals (PEP) 440 [60] den Python Guide zur Versionierung dar. Die PEPs sind als Empfehlung und nicht als Pflicht anzusehen.

3.7.1 Versionierungsformen

Bei der Versionierung von Softwareprodukten unterscheidet man fünf gebräuchliche Arten:

- Die erste Variante der Verisonierung stellt die **Date-of-Release** Versionierung dar. Hierbei wird das Datum des Releases als Versionsnummer genutzt [61]. Dieses kann in verschiedensten Formen wie zum Beispiel „MM.JJ“ oder „Monat Jahr“ umgesetzt werden. Ein Nachteil ist, dass bei dieser Variante keine Aussagen über Art und Umfang der Änderungen des aktuell vorliegenden Releases möglich sind [61]. Neben dem Namen **Date-of-Release** ist auch der Name der Kalender Versionierung[5] in der Literatur gebräuchlich.

- Die **unäre Versionierung** basiert auf dem unären Zahlensystem, welches von den üblichen Zahlensystemen der Informatik (Binär, Oktal, Dezimal und Hexadezimal) abweicht, und im Volksmund als „Bierdeckelnotation“ bekannt ist. Ein Beispiel für dieses untypische Versionierungssystem stellt die Software *TEX*[6] dar, bei der sich die Versionsnummer der stabilen Releases asymptotisch der Zahl Pi nähern.

- Die Verisonierung mit **alphanumerische Codes** ist z. B. bei der Versionierung von Java Software üblich [62]. Hierbei kann neben der numerischen Verisonionierung eine weitere Spezifikation der Version angegeben werden. So ist es üblich herauszustellen, ob es sich um eine Alpha- oder eine Beta-Version handelt. Die Notwendigkeit des tiefergehende Verständnis von Alpha- und Beta-Verison stellt auf der einen Seite einen höheren Grad an Transparenz dar. Sorgt auf der anderen Seite jedoch für eine höheren Einarbeitungszeit, was einen nachteiligen Aspekt der **alphanumerischen Codes** darstellt [62].

- Lenk et al. [63] beschreiben die **fortlaufende Versionierung** nach folgendem Muster „Version.Revision.Variante“. Hierbei stellt die Version die Grundlage zur kontextuellen Einordnung dar. Sie ermöglicht folglich die Identifikation der Vor- und Nachgänger. Eine Revision stellt eine überarbeitete Form und eine Variante eine abgewandekte Form eines Dokumentes darstellt. Daher wird deutlich, dass diese Form der Versionierung primär Dokumenten gilt. Auch wird beschrieben, dass die Arbeit mit **fortlaufender Versionierung** schnell zur Übersichtlosigkeit über den Stand der eigenen Dokumente führen kann, was den Arbeitsfluss zum Stocken bringt.

- Als Standard wird in der gängigen Literaur die **semantische Versionierung** herausgestellt [27, 64]. Dies wird durch die ausdrückliche Empfehlung[7] von Tom Preston-Werner (einer der Gründer von Github) unterstrichen [64]. Die **semantische Versionierung** zeichnet sich unter anderem durch die Konformität mit dem PEP 440 aus.

[5] https://calver.org
[6] https://www.tug.org
[7] https://semver.org

3.7.2 semantische Verisionierung

Aufgrund der Tatsache, dass es sich bei der semantischen Versionierung um den gängigen Standard handelt, soll diese im Folgenden detailierter beleuchtet werden. Die Grundlagenliteratur der sematischen Versionierung stellt die durch Tom Preston-Werner bereitgestellte Beschreibung[7] dar.

Diese Beschreibung sieht das Versionierungslayout „Major.Minor.Patch“ vor, wobei *Major* in diesem Zusammenhang eine Änderung am Softwareprodukt darstellt, die Rückwärtsinkombilitäten verursacht und eine „große“ Veränderung darstellt [64]. Durch Rückwärtsinkompatibilitäten besteht bei den Softwareprodukten, die die vorliegende OSS als Dependency (downstream user [65]) nutzen Handlungsbedarf. Sie müssen ihre Software oder ihre Dateien auf die Form der neuen Version anpassen um diese nutzen zu können. Auch wenn es diese Art der Änderung in der zu versionierenden OSS zu vermeiden gilt, ist sie nicht immer zu verhindern [65].

Minor steht für eine Veränderung am Softwareprodukt, die eine Funktionserweiterung oder eine umfangreiche Revision darstellt, jedoch keine Rückwärtsinkompatibilitäten herbei führt. *Patches* umfassen Fehlerbehebungen sowie kleinere Änderungen [64].

Lam et al. [65] untersuchen in ihrer Publikation die Auswirkungen der beschriebenen Versionierungsform. Sie kommen zu der Erkenntnis, dass die systematische Versionierung eines der Hauptprobleme der Praxis darstellt, da sie zum Beispiel für die Bewertung der Abwärtskompatibilität regelmäßigen Austausch sowie Diskussionen erfordert. Sie heben jedoch auch hervor, dass der Mangel an Informationen, welcher zum Beispiel bei Date-of-Release hervorgehoben wurde bei semantischer Versionierung nicht existiert. Zuletzt lässt sich hervorheben, dass für jede Version neben der Versionsnummer eine Dokumentation (siehe Kapitel 3.9), in Form von Changelogs bzw. Release Notes mitzuliefern ist.

3.7.3 Versionierungssoftware

Die hohe Komplexität des Versionierungsprozesses (siehe Kapitel 3.7.1) sowie der Fakt, dass die Versionierung die Funktionalität der OSS nicht direkt verbessert, führt zu einem Bedarf an softwaregestützten Versionierungsmöglichkeiten. Diese Art der Softwarelösungen wird im Allgemeinen unter dem Begriff der „semantic version calculators“ zusammengefasst [65].

An dieser Stelle wird die Github Action (siehe Kapitel 3.6) *python-semantic-release*[8] für die Versionierung von Pythonpaketen empfohlen. Diese prüft nach jedem Push in die festgelegten zu veröffentlichen Branchs, um welche Art von Änderung es sich handelt, und in welchem Umfang sich die Version des vorliegenden Softwareproduktes verändern muss. Es bietet sich an diesen Prozess mit der Publikation des Softwareprodukts (z. B. auf der Pythonpaketmanagementplatform *PyPI*) zu verknüpfen (siehe Kapitel 3.10 & 4). Vergleichbare CI-Prozesse existieren auch für Repositories mit andere oder gemischten Programmiersprachen.

Die Verwendung der automatisierten semantischen Versionierung impliziert jedoch einige

[8] https://github.com/relekang/python-semantic-release

Anforderungen an die Beschreibung der in das VCS-Repository vorgenommenen Commits der Publizierenden. Durch die oben vorgestellte Github Action wird die Commit Notation von Angular empfohlen, da diese zum einen den Umfang und zudem den Bereich der Veränderung festlegt [66], die sich im wesentlichen aus drei Bestandteilen zusammensetzt (<Typ>(<Geltungsbereich>): <Kurzbeschreibung>):

- Typ – Welche Art der Änderung wurde vorgenommen?
- Geltungsbereich – Welchen Teilbereich des OS-Paketes beinflusst die Änderung?
- Kurzbeschreibung – Was ändert der vorliegende Commit?

Neben den Anforderungen an die Beschreibung eines Commits, benötigt der CI-Prozess zwei weitere Dateien im Repository. Zum einen die *setup.py*, welche die Grundlage für den Installationprozess (siehe Kapitel 3.10) des OS-Paketes darstellt, und somit Versionsnummer und Dependencies definiert. Zum anderen die *setup.cfg*, welche die Attribute der semantischen Versionierung, z. B. *publish to PyPI* oder *Position der Versionsvariablen* festlegt. Ein Beispiel für die aufgelisteten Dateien, sowie dem CI-Prozess wird im Anwendungsbeispiel (siehe Kapitel 4) umgesetzt und kann daher dem SESMG-Repository entnommen werden.

Abschließend lässt sich eine Empfehlung von Taschuk und Wilson herausstellen, die gemäß der Reproduzierbarkeit von Forschungsergebnisse (FAIR-Prinzip) die Veröffentlichung einer OSS-Version zeitgleich mit jeder wissenschaftlichen Publikation empfehlen [27].

3.8 Quellcode Tests

Quellcode Test stärken vier (Qualitäts-)Merkmale einer OSS [67]. Diese vier Merkmale sind

- Richtigkeit,
- Qualität,
- Nutzbarkeit, und
- Wartbarkeit.

Bei Projekten mit abstrakten oder komplexen Problemstellungen ist es sinnvoll zuerst die Tests und anschließend die den Test erfüllende Software zu erarbeiten, man spricht vom sogennanten „Tests first“ Ansatz. Der Vorteil dieser Herangehensweise ist, dass man sich vor der OSS Entwicklung intensiv mit dem Problem auseinandersetzen muss um die Tests zu erstellen. Das schärft das Problemverständnis und kann somit zu einer zielorientierten und effiziernten Softwareentwicklung beitragen [68]. Grundsätzlich unterscheidet man bei Tests zwischen funktionalen Tests und Style Test (z. B. Test auf PEP 8 Konformität) [69]. In der Literatur werden auch andere Eigenschaften zur Kategorisierung der Tests herangezogen [70], die jedoch im Rahmen dieser Arbeit nicht betrachtet werden.

3.8.1 Funktionale Tests

Funktionale Tests dienen der Überprüfung der Funktionalität der Software und zeugen zudem von Qualität und Validität [68]. Dies wird durch die Überprüfung verschieden

großer Softwarebestandteile (Unit, Integration oder System [71]) auf ihr erwartungsgemäßes Verhalten erreicht. In diesem Zusammenhang unterscheidet die Literatur zwischen „Black Box"- und „White Box"-Tests.

Definition White Box Test

> *Bei einem White Box Test haben die Publizierenden vollständige Kenntnis über den Quellcode und können so explizit Schnittstellen und Abschnitte des Quellcodes testen [72].*

Definition Black Box Test

> *Bei einem Black Box Test haben die Publizierenden keine Kenntnis über den Quellcode, sodass hierbei lediglich mit gewählten Eingabeparametern, Rückgabeparameter überprüft werden können [72].*

Unit Tests stellen die kleinste Form des funktionalen Tests dar [68]. Sie testen die einzelnen Funktionen/Methoden unter „Laborbedingungen". Hiermit ist gemeint, dass die Programmierenden vollständige Kenntnis über die an die Funktion/Methode übergebenen Parameter (White Box Test) hat. Und zudem die Untersuchung der Funktion/Methode in einem vom eigenen Betriebssystem entkoppelten Testenvironment stattfindet. Die Programmierenden können somit bei ausreichender Kenntnis der Aufgabe die Zielgrößen (meist die Rückgabewerte) definieren. Der Rückgabewert wird hierbei mit Python-Paketen wie z. B. *pytest*[9] über die Testfunktion assert abgefangen und gegen den Zielwert getestet. Erreicht der Rückgabewert das Zielintervall, so liefert *pytest*[9] nur einen Punkt und färbt bei Erfolg aller Tests die Konsole grün, wodurch sich der Ausdruck „Keep it green!" [58, 68] etabliert hat. Anderenfalls resultiert eine detailliertere Beschreibung des Grundes für den Fehler innerhalb der Tests, mit deren Hilfe das Debugging (Fehlerbehebung) der Ursache ermöglicht wird. Neben den hier beschriebenen positiven Tests, die die Funktionalität der vorliegenden Funktion/Methode validieren, ist es üblich auch sogenannte negative Tests durchzuführen. Diese werden zur Überprüfung des robusten Umgangs mit Fehler die aus falschen Eingabeparametern resultieren herangezogen [27, 73]. Die Verwendung von Testumgebung wie z. B. *pytest*[9] erfordert, dass die Dateien nach dem Schema *test_ <Name der zu testenden Datei>* benannt sind, da sie sonst durch die Testalgorithmen nicht registriert werden können. Im Bereich der funktionalen Quellcode Tests, stellen die Werke von Lenz [68] und Turnquist [74], zwei mögliche Nachschlagewerke für die Unterstützung der Programmierung dar.

Integration Tests testen nicht eine einzelne Funktion/Methode, sondern eine ganze Liste. Sie werden unter White Box Test Bedingungen durchgeführt und überprüfen, ob die logische Verknüpfung zwischen den Funktionen/Methoden funktioniert hat. So lässt sich z. B. prüfen, ob die Übergabewerte wie erwartet gefüllt sind [68]. Ein Problem der Integration Tests ist, dass sie aufgrund ihrer längere Rechenzeit weniger für eine schnelle Kontrolle der Änderung oder des Neuprogrammierten geeignet sind. Dennoch stellen sie den Schritt der Maximierung der Codecoveragerate dar und helfen zudem bei der Identifikation von „totem" – also nicht genutztem oder unerreichbarem – Quellcode [75].

[9] https://github.com/pytest-dev/pytest

Unter der Codecoverage versteht die Literatur den prozentualen Anteil der durch die Tests erreichten Codezeilen.

Die Tests mit dem größten Testumfang sind **System Tests** (auch: Ende zu Ende Tests [76]), die im Gegensatz zu Unit Tests und Integration Tests unter Black Box Test Bedingungen durchgeführt werden. Es wird folglich ohne Kenntnis der Umsetzung der Algorithmuses das Ergebnis einer bestimmten vorher definierten Eingabe überprüft [68]. Eine Kenntnisse über die inneren Zusammenhänge ist bei Tests von solchem Umfang nicht nötig, da sie die vollständige Funktionalität des Softwareproduktes gewährleisten bzw. bestätigen sollen, indem sie bei definierten Eingabeparametern sinnvolle Ergebnisse liefern [68].

Unter dem Begriff **manuelles Testen** versteht man das klassische Ausgeben eines Zwischenergebnisses auf die Konsole, aber auch das Schreiben eines Zwischen- oder Endergebnisses in eine Datei zur visuellen Überprüfung [68]. Zumeist entspricht die visuelle Überprüfung von Ergebnissen einer Plausibilitätsprüfung, die jedoch nur einer Annäherung entspricht und daher nicht der Validierungsgüte der oben beschriebenen Formen von Tests genügt. Häufig wird beim manuellen Testen lediglich nach dem „Black Box“ Prinzip getestet, wodurch das zu lösende Problem nicht vollständig verstanden wird [77].

3.8.2 Software zur Erstellung funktionaler Tests

Ein Beispiel im Bereich der Pythonprogrammierung zur Erstellung von positiven und negativen Tests stellt das Package *Pynguin*[10] dar. Hierbei wird basierend auf der Analyse idealtypischer Pythonprogramme ein Testschema für die jeweiligen Funktionen/Methoden erstellt. Diese müssen in einem zweiten Schritt durch die Programmierenden analysiert und in eine für die Publizierenden und Nutzenden verständliches Format überführt werden. Dies sorgt dafür, dass sie zum einen eine hohe Testcoverage gewährleisten, und zum anderen dem Charakter eines Beispiels gerecht werden, folglich für jede Interessengemeinschaft verständlich/dokumentiert sind [67].

3.8.3 Software zur Berechnung und Erhöhung der Codecoverage

Neben den unterstützenden Softwarelösungen, die die Erstellung der Tests erleichtern sollen (siehe Software zur Erstellung funktionaler Tests), lässt sich auch im Bereich der Tests auf CI-Prozesse zurückgreifen [30]. Diese helfen dabei, die noch nicht durch Tests erreichten Quellcodebestandteile auszumachen. Allgemein gilt, dass eine Testcoverage von mindestens 60 % anzustreben ist. Beispiele für Tools zur Unterstützung bei der Erhöhung der Codecoverage sind:

- Codacy (`https://www.codacy.com`)
- Code Climate (`https://codeclimate.com`)
- Scrutinizer (`https://scrutinizer-ci.com`)
- coveralls.io (`https://coveralls.io`)
- codecov (`https://codecov.io`)

[10] `https://github.com/se2p/pynguin`

3.8.4 Code Style und Linter

Eine weitere Form von Tests, die auf den Quellcode angewandt werden, sind Style Tests bzw. das so gennante Lintering. Hiermit kann die Lesbarkeit des Codes gesteigert und seine Komplexität gesenkt werden [67]. Dies senkt zudem die Hürde zum Einstieg in die Mitentwicklung am Softwareprodukt. Bei Code Styles wird zwischen PEP 8 (dem Code Style der Python Community) und anderen Code Styles wie z. B. dem Google Python Styleguide unterschieden. Demnach sind die jeweilige „Richtlinie“ und die zugehörige erläuternde Literatur als Nachschlagegrundlage heranzuziehen [69, 78].

Der Code Style **PEP 8** hält Konventionen für guten Quellcode fest. So werden zum Beispiel die Tablänge, die Position von Klammern oder auch andere sourcecodespezifische Parameter definiert [69]. Dieser wird im Rahmen dieser Arbeit empfohlen, da er den üblichsten Code Style bei der Pythonprogrammierung darstellt. Eine vollständige Liste kann im Lehrbuch von Browning und Alchin nachgeschlagen werden [79]. Die Style Guide-Vielfalt stellt die Programmierenden vor die Entscheidung, welcher Style genutzt werden soll. Grundsätzlich gilt, basierend auf dem Prinzip des *Clean Code und Testings*, dass sobald sich einmal für eine Stylguide-Konformität entschieden wurde, diese Entscheidung nicht mehr geändert werden sollte [67]. Dies liegt unter anderem daran, dass die Styleguide-Konformität die Programmierung funktionaler Test (siehe Kapitel 3.8.1) vereinfacht [67] und nachträgliche Anpassungen sehr zeitaufwändig sind.

3.8.5 Software zur automatisierten Code Style Prüfung/Anpassung

Die Automatisierung der Code Style Tests wird mithilfe von so genannten Lintern durchgeführt. Diese Tools analysieren den Quellcode der OSS auf Grundlage der im Style Guide festgeschriebenen Eigenschaften von gutem Quellcode und liefern den Programmierenden eine Liste der Verstöße gegen den gewählten Style Guide zurück [67]. Es ist jedoch hinzuzufügen, dass bei diesen Tools, die manuelle Behandlung der durch die Tools gelieferten Verstöße notwendig ist. Beispiele sind in diesem Kontext:

- pycodestyle (`https://pycodestyle.pycqa.org/en/latest/`)
- pylint (`https://pylint.pycqa.org/en/latest/`)
- flake8 (`https://flake8.pycqa.org/en/latest/`)

Aus diesem Nachteil heraus haben sich pre-commit formatter entwickelt. Diese Tools liefern nicht eine Liste an Verstöße, sondern werden „Kollaborierende am VCS-Repository“ in dem sie die Style Guide Konformität durch ihre Kollaboration gewährleisten. Beispiele sind in diesem Kontext:

- autopep8 (`https://github.com/hhatto/autopep8`)
- black (`https://github.com/psf/black`)

3.9 Dokumentation

Mit jedem Softwarepaket ist eine Dokumentation mitzuliefern, welche die Wiederverwendbarkeit, nach FAIR-Prinzip gewährleistet. Hierbei unterscheidet man zwischen der Volltext Dokumentation und der Application Programming Interface (API) Dokumentation. Die Volltext Dokumentation beinhaltet z. B. eine Nutzungs- und eine

Installationsbeschreibung. Die API Dokumentation hingegen detaillierte Informationen über den Quellcode, welche z. B. zur Weiterentwicklung notwendig sind.

3.9.1 Volltext Dokumentation

Die Volltext Dokumentation stellt bei Problemen und Fragen der Nutzenden die erste Anlaufstelle dar [34] und sollte daher nicht vernachlässigt werden [2, 27]. Eine Hürde für die Erarbeitung auführlicher Volltext Dokumentationen ist der Mehraufwand, welcher die Arbeit am Softwareprodukt verlangsamt. Auf der anderen Seite, kann sich dieser Mehraufwand bereits nach geringer Dauer z. B. durch das Vermeiden von Rückfragen durch die Nutzenden rentieren [34]. Die standardisierte Volltext Dokumentation sollte die folgenden neun Kategorien beinhalten [80]:

- Im Abschnitt der **Einleitung**, wird zumeist kurz der Zweck der Software thematisiert. Dies ermöglicht, dass potenzielle Nutzende, die die Dokumentation über eine Suchmaschine gefunden haben, ohne Kenntnis vom zugehörigen VCS-Repositorys, ersichtlich ist wozu die vorliegende Dokumentation gehört. Des Weiteren wird hier der schnellste Weg zur Nutzung des Softwareprodukts beschrieben. Abschließen dient die Einleitung meist der Navigation durch den Rest der Dokumentation, sodass die Struktur für den Lesenden/Suchenden ersichtlich ist [24].
- Die **Contribution Guideline**, umfasst die Regeln zur Kollaboration am Repository, da diese bereits in der *CONTRIBUTING.md* (siehe Kapitel 3.4.2) beschrieben wurden, sollte entweder auf diese verwiesen werden oder alternativ als Volltext in die Dokumentation eingebunden werden. Hierbei sollte jedoch darauf geachtet werden, dass dies nicht mit Copy-und-Paste, sondern über eine automatisierte Einbindung des bereits Geschriebenen geschieht. So kann dem Problem der Up to Dateness in diesem Abschnitt der Dokumentation begegnet werden.
- Unter den Kategorien **Deployment Guideline und Installation Guide** findet man zum einen eine detaillierte Beschreibung des Installationsprozesses (siehe Kapitel 3.10). Und zum anderen eine Beschreibung weitere Deployment Formen, wie z. B. die Nutzung der Software in einem CI-Prozess oder als Dockerimage [81]. Aghajani et al. empfehlen, in diesem Zusammenhang, besonders das Feedback der Nutzenden zu verwenden. Dies rührt daher, dass die Nutzenden meist auf die Schwierigkeiten bei der Installation aufmerksam werden bzw. machen [80].
- Das Anbieten eines Frequently Asked Questions (FAQ)-Bereichs ist in der Softwareverbreitung üblich, und ermöglicht eine schnelle Kommunikation über Probleme bzw. inhaltliche Fragen zwischen Nutzenden und Publizierenden (siehe Kapitel 3.1.1). Das FAQ steht den Nutzenden als Teil der Dokumentation dauerhaft zur Verfügung. Um die Verwendung des FAQ-Bereiches zu vereinfachen, sollten in der Dokumentation einige Regeln definiert werden. Auch könnte ein Template sinnvoll sein, sodass die Suche im FAQ vereinfacht wird.
- Die Abschnitte **How-to und Video Tutorial** reduzieren die Hürde zur Nutzbarkeit soweit wie möglich. Im Vergleich zu einer Text-Dokumentation stellen kurze Videos eine geringere Hürde dar [24, 25]. Hierbei sollte in jedem Tutorial ein abgeschlossenes Problem oder eine abgeschlossene Funktion der OSS dargestellt werden und ein geeigneter Titel bzw. eine geeignete Überschrift gewählt werden.

- Der **Migration Guide** adressiert hauptsächlich die Nutzenden, die das vorliegende Softwareprodukt in ihres eingebunden haben, oder über die vorhandenen Schnittstellen Daten an die OSS übergeben. An dieser Stelle wird beschrieben, welche Änderung die Rückwärtsinkompatibilität (Änderung der ersten Stelle der semantischen Versionsnummer) verursacht hat und welche Maßnahme nun zu ergreifen sind [65].
- Unter den **Release Notes bzw. dem Change Log** versteht man eine Auflistung der Funktionen und Änderungen, die mit der neuen Version / dem neuen Release hinzugekommen sind. Diese werden häufig in einer *CHANGELOG.md* gespeichert, da sie automatisiert aus der Historie der Commits erstellt werden können (siehe Kapitel 3.7). Sie dienen also dem Verständnis der Neuerungen am Softwareprodukt.
- Das **User Manual** stellt in der Regel den Hauptteil der Dokumentation dar, und beschreibt alle Funktionen, die die OSS liefert. So wird zum Beispiel der Aufbau des User Interface (UI) beschrieben, die Form der Eingabedaten oder auch wie die Ergebnisse zu deuten sind. Bei UI wird in diesem Zusammenhang zwischen Graphical User Interface (GUI) und Command Line Interface (CLI) unterschieden, diese sind je nach Festlegung in der SRS zu wählen. Es wird empfohlen Beispieldatensätze zu veröffentlichen, die das Verständnis dieses Kapitels fördern, da sie das Beschriebene verbildlichen [24, 27].
- Der letzte Abschnitt der beschreibenden Dokumentation **Community Knowledge** wird in den meisten OSS-Dokumentation als Sammelstelle für Erkenntnisse des Softwareökosystems (siehe Kapitel 3.1) verstanden. Dies lässt sich durch eine Auflistung der wissenschaftlichen Publikationen, die die OSS für die Produktion ihrer Forschungsergebnisse genutzt haben, aber auch über eine durch die Community fortgeführte „Wissensdatenbank“, die schnelle Problemlösungen oder zentrale Erkenntnisse beinhaltet erreichen.

Problem Up to Dateness:
Meist wird aus zeitlichen Gründen, zwar die Funktionalität der Software erweitert, aber nicht die Dokumentation auf den neusten Stand gebracht [80]. Hierdurch entsteht eine Diskrepanz zwischen Dokumentationsversion und Softwareversion, die aufgrund von mangelnder Transparenz meist nicht ersichtlich wird [34, 80]. Dem Problem kann entgegen gewirkt werden, wenn in der Dokumentation beschrieben wird für welche Version sie gültig ist.

3.9.2 Application Programming Interface Dokumentation

Die Application Programming Interface (API) Dokumentation dient den Publizierenden und beschreibt Funktionalität und Parameter der Klassen, Funktionen und Methoden der vorliegenden OSS. Diese ist als Bestandteil der Volltext Dokumentation aufzufinden (z. B. auf Read the Docs (RTD)) wird jedoch aus den im Quellcode vorliegenden Docstrings erzeugt. Demnach zählen im Wesentlichen zwei Aspekte zur API Dokumentation, zum einen Docstring und Kommentare, die die einfachste Variante der Quellcodedokumentation darstellen, und zum anderen die Tests, die bereits in Kapitel 3.8 behandelt wurden [34, 67].

Ein vollständiger Docstring enthält nach PEP257 [82]

- eine Kurzbeschreibung der vorliegenden Funtkion/Methode,
- eine Liste der Parameter und ihrer Datentypen,
- eine Liste der Rückgabewerte und ihrer Datentypen und
- eine Liste der in der Funktion/Methode enthaltenen Fehlermeldungen sowie ihrer Ursachen.

Hierbei sollte sich bei der Auflistung von Parametern und Rückgabewerten für ein Muster entschieden werden, möglich sind z. B. das Epytext-Muster[11] oder das ReStructuredText (reST)-Muster[12] [34, 65]. Für die Automatisierung von API Dokumentationen mithilfe von Sphinx (siehe Kapitel 3.9.3) wird das reST-Muster[12] empfohlen. Hierbei wird die Docstringdefinition eines Parameters über *:param <Parametername>: <Parameterbeschreibung>* erreicht.

Bei den Quellcodekommentaren liegt die Schwierigkeit darin, die richtige Menge zu finden. Es soll wenn möglich ohne weitere Dokumentation verständlich sein, was die Funktion/Methode erreicht, es soll jedoch nicht jede Zeile kommentiert werden [34, 80]. Hierzu trägt eine sinnvolle Wahl der Variablennamen und eine sinnvolle Funktions-/Methodenstruktur bei (siehe Kapitel 3.11).

3.9.3 Application Programming Interface Dokumentation mit Sphinx

Mithilfe der Software Sphinx können automatisiert, aus den standardisierten Docstrings, API Dokumentationen bereitgestellt werden. Hierzu wird eine *conf.py*, die die Eigenschaften der gehosteten Website – z. B. über RTD – bestimmt im *docs*-Verzeichnis des Repositorys benötigt. In dieser wird festlegt, wo und in welchem Format die dokumentationsrelevanzen Datei vorliegen. Die Inhalte der Dokumentation werden anschließend durch die *index.rst*, die ebenfalls im *docs*-Verzeichnis liegt festgelegt. Für die automatisierte API Dokumentation, muss der Sphinx-Algorithmus an der hierfür vorgesehenen Stelle z. B. über *..automodule:: <Name der Datei>* aufgerufen werden. Sind die notwendigen Einstellungen z. B. Position der *conf.py* im VCS-Repository im Administrationsbereich[13] von RTD festgelegt, so wird die API Dokumentation aus den Docstrings erstellt und in die Dokumentation eingebunden.

3.10 Installation, Dependency Managment und Packaging

Der letzte Aspekt zur Vollständigkeit des Python Packages ist die Installation von Dependencies und das eigentliche Packaging.

Installation

Die Installation nicht-Browser-basierter Software sollte so unkompliziert wie möglich durchführbar sein, da sie sonst bereits zu Beginn abschreckend wirkt [24]. Weiterhin sollte die Anwendung auf allen in der SRS definierten Betriebssystemen installierbar sein. In

[11] http://epydoc.sourceforge.net/epytext.html
[12] https://sphinx-rtd-tutorial.readthedocs.io/en/latest/docstrings.html
[13] https://readthedocs.org/projects/sphinx-rtd-tutorial/downloads/pdf/latest/

diesem Zusammenhang sind Installationsformen über Paketmanager wie *PyPI* oder *make*, aber auch die unüblichere Form des Batchskripts (eine auf dem Rechner ausführbare Installationsdatei) möglich [27]. Allgemein sollte die Notwendigkeit von Adminrechten während der Installation vermieden werden, damit die Zielgruppe nicht eingeschränkt wird [27]. Da *PyPI* den am meisten verwendeten Paketmanager darstellt [83], wird im Folgenden das Packaging spezifisch für *PyPI* betrachtet.

3.10.1 setup.py

Die wichtigste Datei im Kontext der Installation ist die *setup.py*, welche meist im Hauptverzeichnis des Repositorys liegt. Sie legt den Ablauf der Installation bzw. des Setups des Packages fest. Zusätzlich können in der *setup.py* Einstellungen für Testenvironments (siehe Kapitel 3.8) getroffen werden [84].

Die *setup.py* beinhaltet also die Instruktionen für den Algorithmus, der für das Packaging und das anschließende Veröffentlichen auf PyPI verantwortlich ist. Hierunter zählt

- der Name des Pakets,
- eine Kurzbeschreibung,
- eine lange Beschreibung, oder eine Verlinkung auf die Readme (siehe Kapitel 3.4.1),
- die Classifier – Schlagwortliste der Kompatibilitäten z. B. die Pythonversion(en) und
- die Dependencies die für das eigene Softwarepaket notwendig sind.

Um die Dependencies nicht in der *requirements.txt* (die z. B. bei der Lizenzüberprüfung von Nöten ist) und in der *setup.py* aufzulisten, ist es üblich, aus der *setup.py* auf die *requirements.txt* zu verweisen.

3.10.2 Dependency Managment

Die Verwendung von Dependencies ermöglicht die Wiederverwendung des bereits Programmierten und ist daher zur Erfüllung des FAIR-Prinzips erforderlich. Unterstützend wirkt sich der heute übliche Komponenten-basierte Programmieransatz aus [85]. Da es sich bei den Dependencies meist auch um Pakete unter stetiger Weiterentwicklung handelt, hat sich das so gennante Dependency Management, welches plattform-gestützt oder manuell durchgeführt werden kann etabliert. Hierbei steht die Kompatibilität der Pakete im Vordergrund. Dies liegt daran, dass die Kompatibilität in Abhängigkeit der jeweiligen Versionen variieren oder sogar gefährdet sein könnte. Kann diese nicht gewährleistet werden, resultiert eine Nicht-Installierbarkeit des eigenen Pakets. Eine manuelle Überprüfung der Kompatibilität ist fehleranfällig und zeitaufwändig, weshalb an dieser Stelle die Plattform requires.io[14] empfohlen wird. Diese Platform kann über ihren Algorithmus vor Inkompatibilitäten und Sicherheitslücken aus nicht geupdateten Paketen (z. B. über ein Badge) warnen, sodass bei nötigem Handlungsbedarf schnell gewarnt wird [65].

[14] https://requires.io

3.10.3 Software zur Überprüfung der Installation

Auch im Bereich der Installation wird nun der CI-Prozess zur automatisierten Überprüfung des Installationsprozesses betrachtet. Hierin wird zum einen das Betriebssystem (z. B. Linux, MacOS oder Windows) und zum anderen die Pythonversion(en) definiert. Überlicherweise folgt darauf der vorgesehene Installationsprozess, z. B. *pip install -r requirements.txt*. Dies garantiert eine Installationskompatibilität für die in der SRS vorgesehenen Betriebssystem-Python-Kombination und Softwareprodukt.

3.10.4 Software zum Packaging und Hochladen der Software

Konnte im vorherigen Prozess die Funktionstüchtigkeit des Installationsprozesses gewährleistet werden, folgt das Packaging der OSS. Daraufhin kann sie auf eine Paketmanagementsoftware, in diesem Fall *PyPI*, hochgeladen werden. Dies kann auch in Kombination mit dem unter Versionierung (siehe Kapitel 3.7) bereits vorgestellten CI-Prozess zur semantischen Versionierung unter Angabe des PyPI-Tokens erreicht werden.

3.11 Wartbarkeit

Die Wartbarkeit ist während des gesamten Lebenszyklus der OSS relevant, wird jedoch nach dem ersten Packaging von besonderem Interesse. Durch das Entfernen von Duplikaten sowie die Vereinfachung der Funktionen/Methoden gewährleistet sie ein einfacheres Verständnis des bereits Programmierten. Darüberhinaus lässt sich ein Softwareprodukt mit einfacher Wartbarkeit einfacher an die nächste/folgende Forschungsgruppe sowie an die Community weitergeben. Grundsätzlich gilt jedoch bevor Refactoring durchgeführt wird, sollte eine hohe Testcoverage (siehe Kapitel 3.8) gewährleistet sein. Was darauf zurückzuführen ist, dass überprüft werden muss, ob das Refactoring Fehler verursacht hat. Grundsätzlich unterscheidet man vier verschiedene Wartbarkeitstypen [70]:

- adaptive Wartbarkeit (Anpassung an äußere Veränderung)
- präventive Wartbarkeit (Vorsorgliche Verbesserung der Wartbarkeit)
- performante Wartbarkeit (Verbesserung der Performance)
- korrigierende Wartbarkeit (Wartung im Sinne der Fehlerbehebung)

Im Kontext der Wartbarkeit stellt das Buch „Refactoring – Wie Sie das Design bestehender Software verbessern“ [86] die Grundlagenliteratur und somit auch das Nachschlagewerk dar. Refactoring kann entweder automatisiert durch die genutzte Integrated Development Environment (IDE) oder manuell durchgeführt werden.

3.11.1 Duplikate / Code Clones

Kasper und Godfrey führen

- Zeitdruck,
- einfache Wiederverwendung des bereits Programmierten (anderer Softwareprodukt oder auch des eigenen) und
- Lösung eines nicht vollständig verstandenen Problems

als Ursache für Duplikate beziehungsweise Code Clones in Softwareprodukten der FuE, aber auch der freien Wirtschaft, an [87]. Sie beschreiben Duplikate als übliches Resultat des Arbeiten unter Fristendruck. Dennoch sollten diese während des Refactorings entfernt werden, da sie Probleme wie zum Beispiel ungenutzte Codesegmente, Schwierigkeiten bei Les- und Wartbarkeit sowie Fehlerursachen bei Fehlinitialisierung von Variablen mit sich bringen können [87, 88]. Die beschriebenen Probleme stellen zum einen einen Widerspruch zum FAIR-Prinzip dar [3] und sorgen zum anderen dafür, dass die auf die fortlaufende Wartbarkeit (über den gesamten Lebenszyklus) zu investierende Zeit erheblich steigt [88].

In der Literatur wird der Begriff des Duplikates nicht einheitlich genutzt. So wird beispielsweise in der englischsprachigen Literatur häufig der Begriff „Code Clone“ synonym genutzt [87–90]. Im Folgenden wird der Begriff Duplikat verwendet.

Die Beziehung zweier Duplikate kann als reflexive, transitive oder symetrische Äquivalenzbeziehung beschrieben werden, was die Unterteilung von Duplikaten in die folgenden drei Unterkategorien zur Folge hat [88]:

- Typ 1: syntaktisch gleiche (identische, reflexive) Duplikate
- Typ 2: syntaktisch ähnliche jedoch leicht modifizierte Duplikate (transitiv)
- Typ 3: semantisch gleiche (logisch identische, symetrische) Duplikate

In der Literatur werden Duplikate des Typ 2 teilweise noch einmal unterteilt wird [90]. Im Kontext der Duplikate stellt Fowler dar, dass jedem Typ im Refactoring unterschiedlich begegnet werden sollte [86]. Bei einer Redundanz (Typ 1) wird der doppelte Codeblock in eine Methode ausgelagert und an der benötigten Stelle durch einen Aufruf wieder eingebunden (Fowler & Beck: *Funktion extrahieren*) [86, 90]. Das Beheben eines Typ 3-Duplikates ist erheblich komplizierter, da sie zwar logisch das Gleiche bewirken, jedoch in einer anderen Weise programmiert sind. Auch für Behebung von Typ 2/3-Duplikate liefert die Literatur Lösungen [86].

Basierend auf der Klassifikation von Duplikaten wird die Verwendung der Methode nach Schulze und Kuhlemann empfohlen [87, 88]. Hierin wird vom automatisierten Refactoring durch die IDE abgeraten, da dies die Verwendbarkeit des eigenen Quellcodes erschweren kann. Vielmehr wird eine Zwei-Schritt-Methode beschrieben und empfohlen, die den Umgang mit den Ergebnisse einer „Duplication detection“-Software (siehe Kapitel 3.11.3) analysiert und manuell eingearbeitet werden können [89].

3.11.2 Code Smell Analyse

Fowler und Beck [86] beschreiben eine „Code Smell“-Analyse als Maßnahme zur Verbesserung der Wartbarkeit. „Code Smell“ lässt sich in diesem Kontext als Metapher verstehen, da schlechter bzw. ineffizienter Quellcode laut den Autoren einen schlechten Geruch von sich gibt. Die Forschungsarbeit im Kontext der Code Smells hat seit dem Beginn der 2000 Jahre zugenommen [91]. Daher werden in Abbildung 3.4 einige Arten von Code Smells aufgeführt und im Anschluss genauer beschrieben [86].

- **Variablen- und Methodennamen** sollte die Funktionalität bzw. den Inhalt der Selben beschreiben. Wodurch ein Grundverständnis der Variable bzw. der Methode

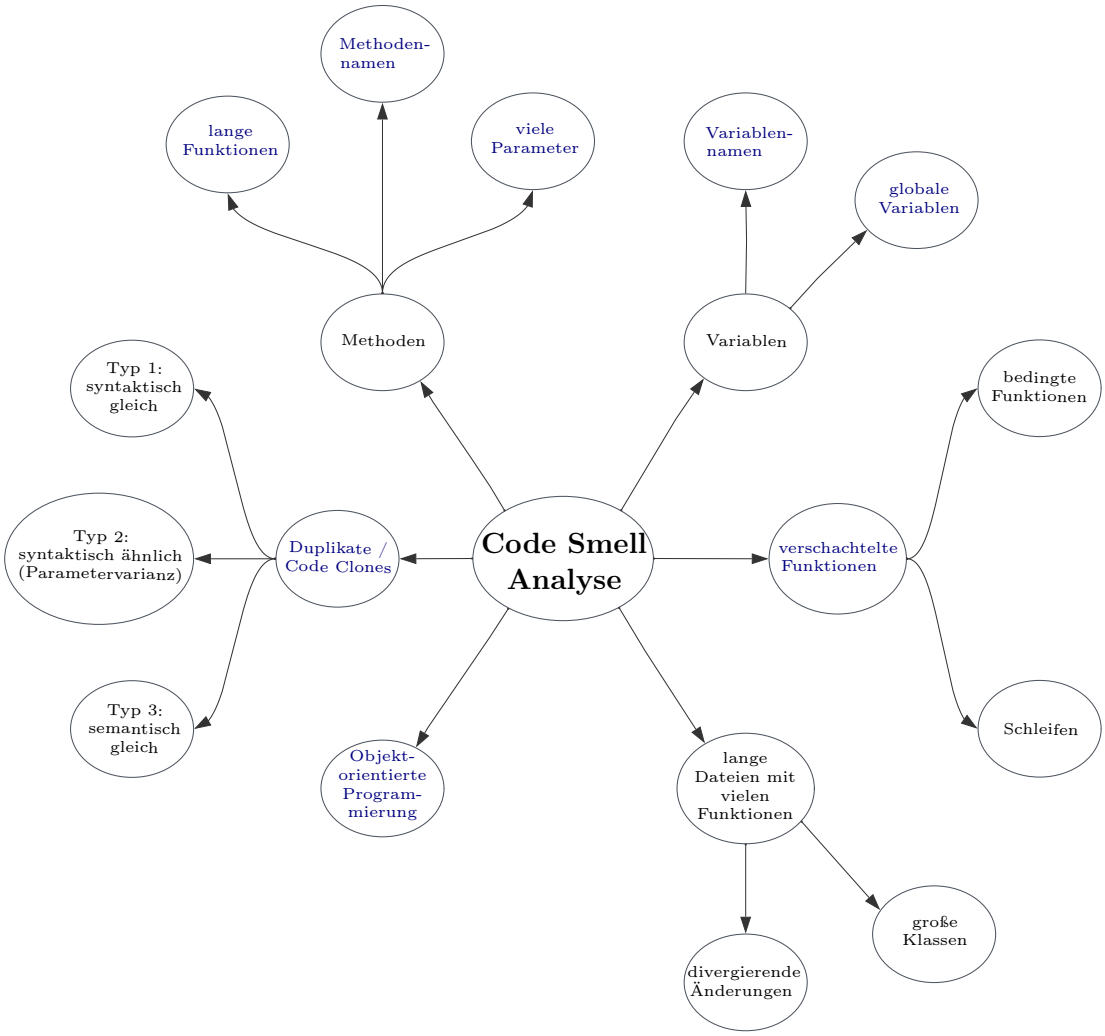


Abbildung 3.4: Code Smell Typen [86] .

aufgrund ihres Names erreicht wird. Fällt es schwer einen geeigneten Namen für Variable oder Methode zufinden, ist dies meist ein Indiz für die Notwendigkeit eines Refactorings (siehe Kapitel 3.11) [86].

- Fowler und Beck [86] beschreiben in diesem Zusammenhang des **Vermeidens langer Funktionen** eine Heuristik: „Verspürt man das Bedürfnis kommentieren zu müssen sollte man eine neue Funktion daraus machen.“ Sie führen des Weiteren drei Vorteile kurzer Methoden an: Sie sind meist
  - selbsterklärend,
  - wiederverwendbar und
  - selektierbar.
- Eine Methode mit **vielen Parametern** ist entweder zu umfangreich oder zu komplex, ein Refactoring (Aufteilen der Methode in zusammenhängende Einzelaufgaben) kann die Les- und Wartbarkeit der Methode erhöhen.

- Mehrere ineinander **verschachtelte bedingte Anweisungen** erhöhen die Komplexität einer Methode und sorgen zusätzlich für eine hohe Ineffizienz des Quellcodes. Sie sollten wenn möglich vereinfacht oder aufgebrochen werden (Fowler & Beck: *Funktion extrahieren bzw. Anweisungen aufteilen*).
- Grunsätzlich sind **globale Variablen** zu vermeiden. Sollte dies nicht möglich sein, so sollten die Variablen als unveränderlich deklariert werden. Das Prinzip der unveränderlichen Variablen (Konstanten) existiert in der dynamischen Programmiersprache Python jedoch nicht.
- **Divergierende Änderungen** sind ein Problem stark vernetzter Softwarestrukturen. Vereinfachend geht es hier hauptsächlich um die Handhabung des folgenden Dilemmas (auch: *Shotgun Surgery*): „Wenn ... passiert müssen im Code die Punkte 1-10 an verschiedenen Stellen verändert werden.“ Dies ist ein Indiz für schwer wartbaren Code, ihm sollte mit einer strikten Modularisierung (auch *Feature-Neid*) Abhilfe geschaffen werden.
- Die Abstraktion des Quellcodes in Klassen (eines der Grundprinzipe der **Objekt orientierten Programmierung**) erfordert meist ein deutlich tieferes Informatikverständnis. Weshalb diese im Kontext der Maßnahmen angeführt, jedoch im Zuge dieser Arbeit nicht weiter thematisiert werden. Detaillierte Infomationen der Objekt orientierten Programmierung in Python, können in zahlreichen Lehrbüchern nachgeschlagen werden, z. B. [92].

3.11.3 Wartbarkeitstools

Die Wartbarkeit wird von Morshed et al. als aufwendigster Bestandteil des Softwarelebenszyklus (siehe Kapitel 3.2 benannt. Um diesen Prozess zu vereinfachen werden Beispiele für unterstützende Tools aufgeührt (siehe Tabelle 3.4). Diese unterscheiden sich hauptsächlich in der Parametrisierung (z. B. die notwendige Länge oder der Prozentsatz der Übereinstimmung eines Duplikates) [87, 90].

Tabelle 3.4: Tools zur Unterstützung des manuellen Refactorings von Python Code.

Name	offline verwendbar	Funktionen
Clone Digger [93] (`http://clonedigger.sourceforge.net`)	✓	Duplikate
Codacy (`https://codacy.com`)	✗	Komplexität, Duplikate, ...
Code Climate (`https://codeclimate.com`)	✗	Duplikate, Code Smells, ...
DuDe (`https://wettel.github.io/dude.html`)	✓	Duplikate Typ 1 & 2
Pylint (`https://github.com/PyCQA/pylint`)	✓	Code Smells, Duplikate, Bugs, ...
Scruitinizer (`https://scrutinizer-ci.com`)	✗	Duplikate, Code Smells, ...

3.12 Zitierbarkeit

Unter der Zitierbarkeit einer OSS versteht die Literatur die Möglichkeit, die OSS in einer wissenschaftlichen Publikation zu zitiern. Dies ermöglicht zum einen die Auffindbarkeit der Software und zum anderen die Reproduzierbarkeit von Forschungsergebnissen (siehe FAIR-Prinzip).

3.12.1 Zenodo

Der schnellste Weg die programmierte OSS zitierbar zu machen, ist die Verknüpfung des VCS-Repositorys mit der Publikationdatenbank Zenodo[15], die jedem Release (siehe Kapitel 3.7) eine Digital Object Identifier (DOI) zuordnet, und den aktuellen Stand des Repositorys abspeichert. Anschließend sollte diese DOI in einer *CITATION.cff* im Hauptverzeichnis des Repositorys abgelegt werden.

3.12.2 Journal Review

Alternativ kann nach Fertigstellung eines Meilensteins, z. B. der ersten standardisierten Version des VCS-Repositorys, eine Publikation der OSS in einem Journal vorgenommen werden. Tabelle 3.5 listet 12 Journals zur Publikation von OSS auf, welche entweder keinen festgelegten Schwerpunkt besitzen, oder ihren Schwerpunkt auf dem Ingenieurwesen haben. Hierbei wird auf die Betrachtung der Anforderungen an den die OSS beschreibenden Artikel, der üblicher Weise Teil der Softwarepublikation ist, verzichtet da sie sich im Guide for Authors nachlesen lassen.

Tabelle 3.5: Auswahl möglicher Journals zur Publikation einer OSS im Bereich des Ingenieurwesen basierend auf `https://www.software.ac.uk/which-journals-should-i-publish-my-software`.

Journalname	Anforderungen & Ziel
Computing in Science & Engineering (CISE)	**Ziel:** Publikation einer Software mit zugehörigen Erkenntnissen von allgemeinem Interesse (Forschungsergebnissen). **Anforderungen:** Einreichung eines Forschungsartikel der die Software nutzt.
Concurrency and Computation: Practice and Experience (CCPE)	**Ziel:** Publikation von Software im Bereich von Big Data, Quantencomputing etc. **Anforderungen:** Nicht definiert.
The Journal of Open Source Software (JOSS)	**Ziel:** Publikation ausschließlich von OSS. **Anforderungen:** • Dokumentation (Funktion & Installation & API) • Tests für die Bestätigung der Funktionalität

[15] `https://zenodo.org`

Journal of Software: Practice and Experience	**Ziel:** Publikation von Software im Bereich IoT, AI, Luft- und Raumfahrt, Energie, etc. **Anforderungen**: Nicht definiert.
Journal of Open Research Software (JORS)	**Ziel:** Forschungssoftware mit hohem Wiederverwendungspotenzial. **Anforderungen:** • prognostiziertes Verhalten der Software entspricht dem tatsächlichen Verhalten • Software in einem geeigneten Repository • offene Lizenz • DOI der Software • Beispiel Ein-und-Ausgabedaten • Dokumentation • Funktionalität auf allen angegebenen Plattformen
Nature Toolbox	**Ziel:** Publikation von Datensätzen. **Anforderungen:** Da aus dem Ziel hervorgeht, dass primär Datensätze gefordert sind, existiert eine hohe Anforderung an deren Validität.
Research Ideas and Outcomes (RIO)	**Ziel:** Offene Publikation von Forschungsideen und -ergebnissen. **Anforderungen:** Aufgrund des offenen Reviewvrfahrens, bei dem die Nutzenden über ihre Rückmeldungen das Review darstellen, sind die Anforderungen nicht definierbar.
SIAM Journal on Scientific Computing (SISC) Software section	**Ziel:** Berechnungsmethoden zur Lösung wissenschaftlicher und technischer Probleme voranzutreiben. **Anforderungen:** Reproduzierbarkeit der Ergebnisse
SoftwareX	**Ziel:** Anerkennung der Auswirkungen von Software. **Anforderungen:** OSS mit Supportmaterial (Dokumentation & Beispielen)
Advances in Engineering Software	**Ziel:** Kommunikation der Fortschritte in computerbasierten Ingenieurtechniken. **Anforderungen:** Nicht definiert.

Coastal Engineering	**Ziel:** Publikation von Forschungsinhalte und Softwareprodukte der Küstenforschung. **Anforderung:** Nicht definiert.
Renewable Energy	**Ziel:** Themen und Technologien erneuerbarer Energiesysteme und -komponenten. **Anforderungen:** Nicht definiert.

Bei genauerer Betrachtung der zu Anfang 12 Journals, konnten bereits 2 aufgrund sehr spezifischer Schwerpunkte (Coastal Engineering) bzw. der Notwendigkeit eines zusätzlichen Forschungsartikels (Computing in Science & Engineering) ausgeschlossen werden. Zudem fällt auf, dass bei den Journals, die neben ihren üblichen Publikationen auch Softwarepublikation ermöglichen, die Anforderungen häufig intransparent bleiben. Im Rahmen der Betrachtung wiesen besonders die Journals mit einer thematischen Fokussierung der Publikation von OSS zum einen das The Journal of Open Source Software (JOSS) und zum anderen das Journal of Open Research Software (JORS) einen hohem Grad an Transparenz auf, weshalb diese im Rahmen dieser Arbeit für eine OSS-Publikation empfohlen werden.

4 Diskussion SESMG

Nachdem nun die theoretische Grundlage für die Standardisierung eines Python Software Paketes beschrieben wurde, soll basierend auf dem SESMG eine Überprüfung der Anwendbarkeit des Leitfadens vorgenommen werden. Hierbei wird von der klassischen Struktur – den deskriptiven und den diskutierenden Teil strikt zu trennen abgesehen, und Schrittweise der Ablauf des Leitfadens vollzogen. Sollte die Durchführung einzelner Aspekte im Rahmen der vorliegenden Arbeit nicht möglich gewesen sein, so werden Best Practice Beispiele angeführt, die eine korrekte Umsetzung der im Leitfaden dargestellten Zusammenhänge abdecken.

4.1 Status Quo

Die OSS wurde zu Beginn im Rahmen einer Masterarbeit auf dem FH Münster betriebenen Gitlab Server[16] veröffentlicht. Nach einiger Zeit der Weiterentwicklung, entschieden sich die Publizierenden (derzeit 9 Personen) für das nicht-institutionsbezogene VCS Github [14], sodass fortan die Auffindbarkeit für alle Nutzenden möglich wurde. Neben der Veröffentlichung auf Github, wurden einige Releases (über Zenodo) [14] zitierfähig gemacht. Der Sourcecode ist dem Draftzustand zuzuordnen, da zwar zu einem hohen Anteil kommentiert wurde und Docstrings vorliegen, jedoch bis dato weder ein besonderes Augenmerk auf Wartbarkeit (Codeclimate Note D mit einer Einarbeitungsdauer von 4 Monaten) noch auf automatisierte Tests gelegt wurde (Coveragerate war aufgrund fehlender Testinfrastruktur nicht bestimmbar). Weiterhin lässt sich hinzufügen, dass die Dependencies des Repositorys zu einem hohen Anteil mit meist veralteten Versionen fixiert wurden. Dies sorgt zwar für eine Erleichterung des Supports, schließt jedoch auch aus, dass die Weiterentwicklungen der Dependencies genutzt werden konnten. Für den SESMG liegen eine Vielzahl an Beispielen vor, die jedoch zu einem großen Teil unzureichend erläutert sind [94]. Des Weiteren existiert eine umfangreiche Volltextdokumentation [95], diese beschreibt erstens die Funktionalität des vorliegenden Softwareprodukts, zweitens eine hohe Anzahl der Parameter des für die Modellierung und Optimierung auszufüllenden Kalkulationsdatenblattes und drittens einen großen Teil der API, da diese bereits automatisiert aus den Docstrings über Sphinx [96] erstellt wird. Ein aktueller Stand des Repository auf Grundlage der Badges ist in Abbildung 4.1 zu finden.

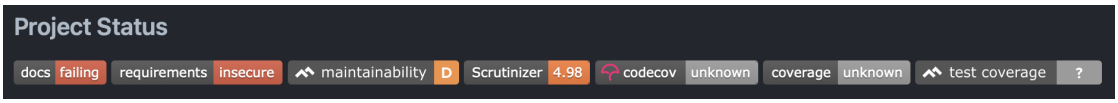


Abbildung 4.1: Status Badges des SESMGs im Status Quo.

[16] https://git.fh-muenster.de

4.2 Softwarespezifikationsphase

Die Softwarespezifikationsphase des SESMGs wurde nicht wie im Leitfaden empfohlene basierend auf einer SRS erarbeitet. Dennoch wurde sie im Rahmen einer ersten Studie [97] und einer Masterarbeit durchgeführt. Hierbei wurden die notwendigen Funktionen der Software definiert, und anschließend aufgrund dieser Anforderungen die Suche nach einem Framework und die Programmierung des SESMGs begonnen.

Da eine standardisierte Form der SRS für den SESMG nicht vorliegt, wird an dieser Stelle auf eine Beispiel SRS[17], die zum Zwecke der Lehre enstanden ist verwiesen. Diese genügt der von Institute of Electrical and Electronics Engineers (IEEE) beschriebenen Struktur (siehe Kapitel 3.2).

Aufgrund der beschriebenen Eigenschaften wird der SESMG, auch wenn keine standardisierte Form der SRS vorliegt beim Einstiegspunkt 3 verortet. Entspricht daher einer neu zu programmierenden Software mit bereits bestehendem VCS-Repository.

4.3 Lizensierung

Im Status Quo wurde das vorliegende VCS-Repository unter der MIT-Lizenz betrieben und bereitgestellt. Wie in Kapitel 3.5.2 beschrieben wurde, wurde mit der Software *LicenseCheck* eine Überprüfung der Lizenzkompatibilität durchgeführt. Hierzu musste in einem ersten Schritt die noch nicht vorhandene *requirements.txt* erstellt werden. Abbildung 4.2 zeigt die Ergebnisse der Lizenzüberpürfung. Hierbei stellte sich heraus, dass die Fremdsoftware *richardsonpy*[18] – die verwendet wird um stochastisch Lastgänge zu erzeugen – unter GNU GPL 3.0 veröffentlicht wurde und es somit eine Unverträglichkeit zwischen der Dependencylizenz und der selbstgewählten Lizenz gab. Da die Fremdbibliothek jedoch für die Funktion des SESMG notwendig ist, wurde die Lizenz des SESMG auf GNU GPL 3.0 geändert.

Compat	Package	License
✅	basemap	MIT License
✅	dash	MIT License
✅	dash_canvas	MIT
✅	demandlib	MIT
✅	dhnx	
✅	graphviz	MIT License
✅	memory-profiler	BSD License
✅	oemof.solph	MIT License
✅	oemof.thermal	
✅	open_fred-cli	BSD License
✅	openpyxl	MIT License
✅	osmnx	MIT License
❌	richardsonpy	GNU General Public License v3
✅	scikit-learn-extra	new BSD
✅	seaborn	BSD License
✅	sympy	BSD License
✅	xlsxwriter	BSD License

Compat	Package	License
✅	basemap	MIT License
✅	dash	MIT License
✅	dash_canvas	MIT
✅	demandlib	MIT
✅	dhnx	
✅	graphviz	MIT License
✅	memory-profiler	BSD License
✅	oemof.solph	MIT License
✅	oemof.thermal	
✅	open_fred-cli	BSD License
✅	openpyxl	MIT License
✅	osmnx	MIT License
✅	richardsonpy	GNU General Public License v3
✅	scikit-learn-extra	new BSD
✅	seaborn	BSD License
✅	sympy	BSD License
✅	xlsxwriter	BSD License

Abbildung 4.2: Ausgabe der Software LicenseCheck unter Verwendung der MIT-Lizenz (links) und der GNU GPL 3.0 Lizenz (rechts).

[17] https://github.com/mahmoudai1/school-management-system/blob/main/SRS.pdf
[18] https://github.com/RWTH-EBC/richardsonpy

4.4 Readme

Eine REAMDE lag bereits im Status Quo des VCS-Repositorys des SESMG vor. Diese lag abweichend von dem im Leitfaden empfohlenen Markdownformat im ReStructured-Text Format im Hauptverzeichnis vor. Sie beinhaltete jedoch eine Kurzbeschreibung (Was?), eine Verlinkung zu Beispielen und zur Dokumentation (Referenzen), eine Möglichkeit der Kontaktaufnahme (Wer?) und Informationen über das finanzierende Forschungsprojekt „R2Q – RessourcenPlan im Quartier“.

Im Zuge der Standardisierung wurde

1. das Dateiformat der README geändert,
2. die fehlenden Aspekte wie zum Beispiel der Installationprozess hinzugefügt,
3. Badges wie „What“ oder „Why“ für das einfachere Auffinden eingebunden und
4. die README um die im Status Quo noch nicht existenten Dateien des *CODE_OF_CONDUCT.md* und der *CONTRIBUTING.md* ergänzt.

4.5 Contributing

Eine *CONTRIBUTING.md* war im Status Quo noch nicht Bestandteil des Repositorys. Vielmehr wurde die Nutzenden, aufgrund der noch händelbaren Anzahl, immer persönlich instruiert, was und wenn ja wie sie am vorliegenden Repository teilhaben können. Auch Issue Templates[19] und PR Templates[20] wurden im Rahmen der Standardisierung dem Repository des SESMG hinzugefügt.

Es bleibt jedoch abzuwarten, wie die Issue und PR Templates durch die Nutzenden angenommen werden. Best-Practice Beispiele, wie zum Beispiel die OpenPIV[21] zeigen, dass die Templates meist Anwendung bei der Community erreichen. Zusätzlich ist hinzuzufügen, dass aufgrund des Mangels an Test (siehe Kapitel 4.7), die Standardisierung der *CONTIRBUTING.md* noch nicht abgeschlossen werden konnte, was zugleich Einfluss auf die Beantwortung der dritten Forschungsfrage hat (siehe Kapitel 5 & 6).

4.6 Versionierung

Der Master-Branch des SESMGs wurde zuletzt am 21.09.2021 als Version 0.2.0 veröffentlicht. Seither fanden in diesem Branch circa 125 Commits (Stand vor Beginn der Standardisierung) statt, die zum einen Fehler behoben und zum anderen die Dokumentation aktualisiert haben. Des Weiteren wurde bei der Beschreibung der Commits meistens darauf geachtet, dass diese verständlich macht was im Commit verändert wurde. Dennoch zeigt sich auch, dass bei schnellem oder häufigen Pushen die Qualität der Commitbeschreibungen bis hin zu „Updating <Name der geänderten Datei>“ abgenommen hat. Es hätte mehrfach zu einer Erhöhung der letzten Stelle der Version (Patch) kommen müssen (siehe Kapitel 3.7).

[19] https://github.com/chrklemm/SESMG/tree/master/.github/ISSUE_TEMPLATE
[20] https://github.com/chrklemm/SESMG/blob/master/.github/PULL_REQUEST_TEMPLATE
[21] https://github.com/OpenPIV/openpiv-python

Ferner wurde im Rahmen dieser Arbeit der unter Kapitel 3.7.3 beschriebene CI-Prozess[22] hinzugefügt, der fortan die semantische Versionierung des Master-Branchs automatisiert vornehmen wird. In diesem Zuge wurden die Publizierenden über eine Beschreibung des Angular Commit Schema in der *CONTIRBUTING.md*[4.5] aufmerksam gemacht.

Die Erstinbetriebnahme des CI-Prozesses verlief jedoch nicht fehlerfrei. Die bereits existierenden Releases stellten eine Hürde zur Verwendung des CI-Prozesses dar. Dieser registrierte die bereits existierenden Releases nicht als Releaseversion, weshalb über Erstellen später nicht mehr existenter Releases suggeriert werden musste, dass die Version des Repositorys sich fortlaufend erhöht.

4.7 Quellcode Tests

4.7.1 Funktionale Tests

Die Testcoverage des SESMG konnte im Status Quo nicht ermittelt werden, da keine Quellcodetest existierten. Im Rahmen dieser Arbeit wurde in einem Fork zu Testzwecken ein erster Unit Test[23] implementiert, der eine Bestimmung der Testcoverage ermöglichte. Diese beträgt je nach CI-Tool bei 10 - 14 %[24][25]. Die Testcoverage resultiert jedoch abgesehen von dem ersten implementierten Unit Test ausschließlich aus der Überprüfung der Funktions-/Methodenköpfe sowie der Importstatements.

Bei der Anwendung der in Kapitel 3.8.2 beschriebenen Software zur Unterstützung bei der Erarbeitung von Tests, konnte diese jedoch nicht wesentlich erhöht werden. Dies ist darauf zurückzuführen, dass die Parameter der Funktionen/Methoden meist keine primitiven Datentypen sind und somit die automatisierte Erstellung von Tests erschweren bzw. verhindern.

Aufgrund des zeitlichen Rahmens der vorliegenden Arbeit, ist die Konzeptionierung und Programmierung von funktionalen Tests für den SESMG nicht möglich und wird daher als Auftrag in die Tätigkeit der Publizierenden übertragen (siehe Kapitel 6). Als Best-Practice-Beispiel für funktionale Tests mit komplexen Datenstrukturen lässt sich *oemof-solph*[26] nennen. Dort wird mit einer Testcoverage von 96 - 98 % (abhängig vom testenden Tool) eine nahezu vollständige Überprüfung des Softwarepaketes erreichen. Herauszustellen ist, dass es sich hierbei nahezu ausschließlich um positive Tests handelt, sodass die robuste Behandlung falscher Eingaben teilweise unzureichend überprüft wird.

4.7.2 Style Tests

Im Rahmen dieser Arbeit wurden erfolgreich CI-Prozesse zur Code Style Formatierungen[27] eingeführt. Sodass fortan automatisiert eine PEP 8 Konformität des im Master-

[22] https://github.com/chrklemm/SESMG/blob/v0.3.0/.github/workflows/semantic_versioning.yml
[23] https://github.com/GregorBecker/SESMG/blob/dev_open_district_upscaling/tests/test_district_heating.py
[24] https://coveralls.io/github/GregorBecker/SESMG?branch=dev_open_district_upscaling
[25] ,https://app.codecov.io/gh/GregorBecker/SESMG
[26] https://github.com/oemof/oemof-solph
[27] https://github.com/chrklemm/SESMG/blob/master/.github/workflows/black.yml

Branch liegenden Quellcodes gewährleistet werden kann.

4.8 Dokumentation

Eine Volltext Dokumentation des SESMGs lag bereits im Status Quo des VCS-Repositorys vor. Hierin wurde erstens erklärt, wie die Modellierung der Energiesystemkomponenten auf Grundlage der Graphentheorie abgebildet werden. Zweitens welche Eingabeparameter in das für die Simulation und Optimierung notwendige Kalkulationsdatenblatt eingehen. Und drittens welche Funktion welcher Teil der GUI abdeckt. Zudem wurde bereits eine mit *Sphinx automodule* (siehe Kapitel 3.9.3) betriebene API Dokumentation erstellt.

Im Status Quo konnte aufgrund von Inkompatibilitäten der Dependencies (siehe Kapitel 4.9) keine Dokumentation mehr erstellt werden (siehe Abbildung 4.1 - build: failed). Da diese im Rahmen dieser Arbeit behoben wurden, ist die Erstellung der Dokumentation wieder möglich. Bei Betrachtung der Dokumentation wird jedoch nicht ersichtlich, zu welche Version des SESMG diese gehört. Es besteht folglich das Problem der Up-to-Dateness (siehe Kapitel 3.9). Ein Beispiel für eine standardisierte Dokumentation stellt der Python Developer Guide[28] dar.

4.9 Installation und Packaging

Im Bereich der Installation wurden im Status Quo, drei Installationsdateien für die in der Spezifikation geplanten Betriebsysteme Linux, MacOS und Windows bereitsgestellt. Alle beinhalteten die Dependencies, die für die Installation des SESMGs notwendig sind.

Auffällig ist in diesem Zusammenhang zum einen die Redundanz, der mit der Erstellung einer *requirements.txt* begegnet wurde. Zum anderen fielen die fixen Versionsnummern auf, die die genutzten Dependencies auf einer veralteten Version[29] (Requires.io: 2 unsichere und weitere Veraltete Dependencies) hielten. Dies diente der Verringerung des Aufwandes der notwendig ist, die Kompatibilität zu gewährleisten, widerspricht jedoch dem in Kapitel 3.10 beschriebenen Standards.

Des Weiteren wurden die Dependencies betrachtet und abgewogen, ob diese für die Funktionalität der Software von Nöten sind. Allgemein gilt, dass für eine gute Wartbarkeit, die Anzahl der Dependencies so gering wie möglich zu halten ist [27]. Das Anpassen der Dependencies, welches im Rahmen dieser Arbeit begonnen wurde, war nicht problemfrei lösbar. So wies ein Fremdpaket in seiner neueren Version *geopandas* als für die Installation notwendiges Paket aus. *Geopandas* ist jedoch problematisch, da es sich auf windows-betriebenen Rechnern nur manuell installieren lässt und daher den für den SESMG üblichen Prozess unterbrach. Dennoch konnte der Status der Dependencies in der Bewertung nach Requires.io (siehe Kapitel 3.10) auf „up-to-date“[30] verbessert werden.

[28] https://devguide.python.org/
[29] https://requires.io/github/chrklemm/SESMG/requirements/?tag=v0.2.0
[30] https://requires.io/github/chrklemm/SESMG/requirements/?branch=master

Die Veröffentlichung auf dem Paketmanager „PyPI“ bot sich im Kontext der Standardisierung des SESMGs nicht an. Dies liegt daran, dass dieser nicht den klassischen Charakter eines Pythonpackages, welches über CLI Eingaben oder über die Einbindung des Codes in ein eigenes Softwareprodukt aufweist. Vielmehr handelt es sich beim SESMG um eine Applikation mit einer GUI. Bei der Publikation über einen Paketmanager, müsste sich diese nach Installation über das Python import Statement oder über einen zusätzlichen Methodenaufruf öffnen, was der Charakteristik des SESMGs (Nutzung ohne Kontakt zur Programmiertsprache Python) entgegen stehen würde. Wird dennoch eine Paketlösung angestrebt, kann alternativ eine Vorbeitung für einen Docker[31] veröffentlicht werden. Dies benötigt jedoch auf Rechnern mit einem anderen Betriebssystem als Linux die Installation von Docker[31]. Im Rahmen dieser Arbeit wurde eine Dockerumgebung[31] für den SESMG erstellt, die jedoch aufgrund des derzeitigen GUI-Betriebs mit *tkinter* nicht auf allen Betriebssystemen nutzbar ist. Eine andere GUI, die die Kompatibilität mit allen festgelegten Betriebssystemen unterstützt befindet sich in der Entwicklung. Daher wurde der Installationsprozess vorerst nicht verändert sondern lediglich aktualisiert.

4.10 Wartbarkeit

Auch wenn die Abdeckung der Tests im Status Quo des SESMGs nicht gewährleistet war (siehe Kapitel 4.7), konnte aufgrund der umfangreichen manuellen Testbarkeit des Produktes, eine Verbesserung der Wartbarkeit hervorgerufen werden. So wurden z. B. die sich noch im Repository des SESMG befindlichen Dependencies entfernt und durch die Teilhabe an den fremden Repositories[32][33] die Möglichkeit geschaffen, diese ohne Warnungen und mit voller Funktion wieder über den Paketmanager „PyPI“ zu nutzen. Die in Kapitel 3.11 aufgeführte Software Code Climate vergab dem SESMG zu Beginn die Note D. Hierbei fielen einige Duplikate auf, die wie in Kapitel 3.11 aus dem Fristendruck resultieren. Des Weiteren wurden auch einige Code Smells, wie z. B. Hilfsvariablen mit unverständlichen Namen vor gefunden. Ferner konnte an der Anzahl der Kommentare, die für die Entwicklung des jeweiligen Softwareabschnitts verfügbare Zeit abgeleitet werden. Mussten Bestandteile in wenigen Tagen programmiert werden, so wies der Quellcode wenige Kommentare auf, wurde er schon häufiger gesichtet und diskutiert waren die Kommentare für die Erläuterung der Funktion ausreichend. Den im SESMG vorhandenen Duplikaten, die meist von Typ 2 (syntaktisch ähnlich) waren, wurde mit der Auslagerung in eine Funktion begegnet. Diese wurde in einem zweiten Schritt an den notwendigen Stellen wieder eingebunden. Aufgrund der vorgestellten Maßnahmen, konnte der Umfang des VCS-Repositorys von zu Beginn circa 24 000 Zeilen Quellcode, zu jetzt circa 12 000 Zeilencode reduziert werden (siehe Abbildung 4.3). Abschließend lässt sich festhalten, dass auch eine Modularisierung auf kleinere Quellcode-Dateien durchgeführt wurde, die von nun an die Wartbarkeit des VCS-Repositories fördern wird.

[31] https://www.docker.com
[32] https://github.com/oemof/feedinlib/pull/73
[33] ,https://github.com/oemof/demandlib/pull/51

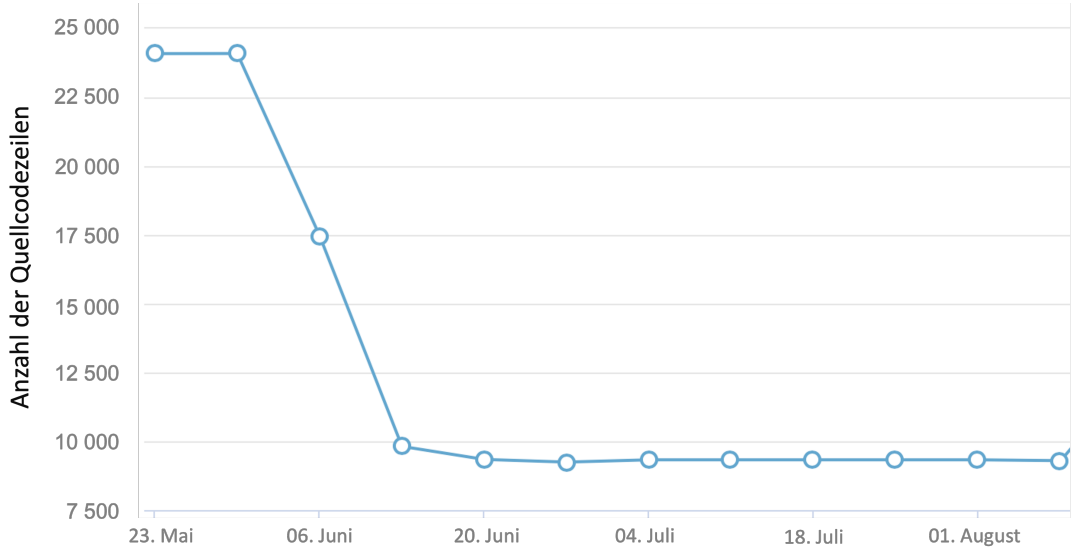


Abbildung 4.3: Entwicklung des Codeumfangs des SESMGs.

4.11 Zitierbarkeit

Der SESMG wurde bereits im Status Quo bei Zenodo gelistet, es wurde jedoch erst im Zuge der Standardisierung die Repositorydatei[34] zur Verlinkung der Zitierbarkeit im VCS erstellt.

4.12 Bewertung der Anwendbarkeit

Die Anwendung des in Kapitel 3 erarbeiteten Leitfaden konnte im Rahmen der vorliegenden Arbeit begonnen werden. Dennoch musste an verschiedenen Stelle (z. B. Einordnung in den Leitfaden, Verisionierung) eine Lösung für die durch bereits im Vorfeld vollzogenen Handlungen aufkommenden Probleme gefunden werden. Demnach ist die Anwendung des Leitfadens bei einer bereits bestehenden Software komplizierter als bei einer neu zu erstellenden. Nichts desto trotz ist der zeitliche Ablauf, der im Leitfaden basierend auf dem Softwarelebenszyklus (siehe Kapitel 3.2) erarbeitet wurde umsetzbar impliziert jedoch, dass die Prozessschritte vollständig abgeschlossen sind bevor mit dem folgenden Prozessschritt fortgesetzt wird.

[34] https://github.com/chrklemm/SESMG/blob/master/CITATION.cff

5 Fazit

Viele Softwareprodukte der Open Source (OS)-Entwicklung verbleiben im nicht standardisierten Zustand. Dies lässt sich meist auf die unzureichende Kenntnis und den Zeitdruck in der Entwicklung von OSS zurückführen. Dies gilt insbesondere inm Bereich der FuE. Auch sind fehlende Finanzierung und die übliche Übergabe von Wartungs-/Instandhaltungsaufwand sowie des Supports an die Community Konkurrent der Standardisierung einer OSS. Wird eine Software gar nicht veröffentlicht, oder ist für die Nutzenden nicht verständlich, so müssen diese eine neue Lösung programmieren, was zum einen ineffizient und zum anderen auch entgegen dem in der FuE üblichen Findable, Accessible, Interoperable and Reusable (FAIR)-Prinzip ist.

Der in dieser Bachelorarbeit erarbeitete Leitfaden stellt die **Schritte zur Standardisierung** einer OSS dar. Er hilft dabei dem Problem der nicht standardisierten OSS entgegen zu wirken. Dies ist darauf zurückzuführen, dass dieser mithilfe der empfohlenen unterstützenden Software eine schnelle und weiterreichende Schritt-für-Schritt Lösung zur Verbesserung des OSS-Standardisierungszustandes darstellt.

(Nicht-)funktionalen Anforderungen der unterschiedlichen Interessengemeinschaften werden entweder in der Kommunikation über die in Kapitel 3.1.1 beschriebenen Formen, über die Dokumentation oder über andere Schritte des Leitfadens erfüllt. Der größte **Konflikt**, der im Bezug auf die **(nicht-)funktionalen Anforderungen** der Interessengemeinschaften auffiel, ist der Konflikt zwischen Fristendruck und Standardisierung des Softwareproduktes für Nutzende. Was darauf zurückzuführen ist, dass die Publizierenden unter Fristendruck diese außer Acht lassen.

Die **Publikationsfähigkeit** einer Software, lässt sich aus zwei Blickwinkel beantworten, zum einen ist eine zitiertfähige Publikation z.B. auf Zenodo bereits zu empfehlen bevor die Software standardisiert wurde. Hierbei gilt der Grundsatz, dass eine Veröffentlichung der Software so früh wie möglich angestrebt werden sollte. Zum anderen gilt, dass wenn eine Publikation in einem Journal für OSS angestrebt wird, nach der umfassenden Standardisierung des VCS-Repositorys, aus Sicht der OSS hierzu keine Hürde mehr vorhanden ist.

Bei der Überprüfung der Anwendbarkeit anhand des SESMGs konnten einige Aspekte wie z. B. die automatisierte Anpassung des Quellcodes auf die Code Style Konformität umgesetzt werden. Dennoch kam es bei der Implementierung der automatisierten Versionierung zu einem Konflikt zwischen bereits bestehenden Versionen und den neuen durch den Algorithmus zu schaffenden Versionen. Auch musste im Bereich der Lizenz ein Wechsel durchgeführt werden, was zum einen auf die mangelnde Kenntnis und zum anderen auf die fehlende Zeit zurückzuführen.

Daher lässt sich allgemein gültig empfehlen, die Standardisierung von OSS so früh wie

möglich und so automatisiert wie möglich zu beginnen. Um somit dem erheblichen Mehraufwand der nachträglichen Standardisierung zu entgehen.

6 Ausblick

Im Kontext des Anwendungsbeispiels SESMG, konnte die Standardisierung des Repositorys im Rahmen dieser Arbeit maßgeblich verbessert jedoch nicht abgeschlossen werden (siehe Kapitel 4). Diese wird im Anschluss an diese Arbeit fortgeführt, sodass eine Publikation im JOSS oder im JORS möglich wird. Hierfür müssen vorallem Unit Tests, die für die Journals ein notwendiges Kriterium darstellen, erarbeitet werdem. Zusätzlich wird wie in Kapitel 3.12 beschrieben eine Beschreibung der Funktionalität der Software gefordert, die die Publizierenden, zur Planung und Erstellung der noch fehlenden SRS bewegen wird. Daraufhin kann diese im weiteren Entwicklungsprozess als Mittel des QM angesehen werden.

Danksagung

An dieser Stelle möchte ich meinen Dank an alle die aussprechen, die mich bei der Erarbeitung meiner Bachelorarbeit unterstützt haben. Besonders möchte ich an dieser Stelle den Dank an Prof. Dr.-Ing. Peter Vennemann und Christian Klemm richten, die mir im Rahmen meiner Praxisphase tiefere Einblicke in die Tätigkeit in der Forschung gewährten und mir die Erarbeitung der vorliegenden Arbeit ermöglichten. Des Weiteren möchte ich mich für die Diskussionen über die Inhalte der vorliegenden Arbeit bei Christian Klemm, Janik Budde und Jan N. Tockloth bedanken, die mich bei der Erstellung des Leitfadens maßgeblich voran brachten. Abschließend bedanke ich mich bei Dr. Jannik Hüls für die Betreuung bei der vorliegenden Arbeit.
Der im Leitfaden behandelte SESMG wurde im Forschungsprojekt „R2Q – RessourcenPlan im Quartier“ [16] (Förderkennzeichen 033W102A) stetig weiter entwickelt und prägte in diesem Zusammenhang maßgeblich meine praktische Tätigkeit.

Glossar

Badge
Badges sind kleine Schilder, die verschiedene Eigenschaften in der Readme eines Repositorys abbilden können. So kann z. B. angezeigt werden, ob die Dependencies up-to-date sind oder die Dokumentation weiterhin funktionstüchtig ist [98].

Branch
Ein Branch ist eine parallele Version des Repositorys, die sich unabhängig vom Orginal verändern und später wieder zusammführen lässt [99].

Commit
Ein Commit beschreibt eine Änderung an einer oder mehrerer Dateien des Repositorys. Diese Erhalten bei der Erstellen eine einzigartige ID, die die Änderungen rückverfolgbar macht [99].

Dependency
Fremdpakete, die für die Funktionalität des betrachteten Softwareprodukts notwendig sind [99].

Docstring
Erster Ausdruck einer Klasse, Funktion/Methode oder eines Moduls, der die Parameter sowie die Funktionalität beschreibt [100].

Issue
Im Reiter Issues des Repositorys, können die Nutzenden Vorschläge für Verbesserungen sowie Fragen zum Repository an die Community stellen [99].

merge
Das Mergen beschreibt den Prozess des Zusammenführen zwei veschiedener Repositorystatus [99].

PR
Ein Pull Request (PR) stellt den Antrag auf Einpflegung Änderungen eines Nutzenden in seinem Fork in die Struktur des Repositorys dar. Sie kann durch die Publizierenden überprüft und daraufhin angenommen oder abgeleht werden [99].

Pushen Das Pushen meint das Senden der Änderungen an einem Repository an den Server von Github [99].

Quellcode
„Ein Quellcode ist ein von Menschen geschriebener Text, den Maschinen ausführen können.“ (englisch: Source code) [101].

Refactoring
Umstrukturierung und Effizienzsteigerung des Softwareprodukts ohne Veränderung an der Funktionalität [86].

Release
Ein Release ist der von Github zur Verfügung gestellte Weg des Packagings und der Verbreitung des Softwareproduktes an die Nutzenden [99].

Repository
Ein Repository ist das grundlegendste Element von GitHub und entspricht der Projektordnerstruktur [99].

Literatur

[1] Matthias Emmanuel Stürmer und Jasmin Myriam Nussbaumer. *Open Source Studie Schweiz 2021*. Techn. Ber. Universität Bern, Juli 2021. DOI: 10.48350/157326. URL: https://boris.unibe.ch/157326/ (besucht am 01. 06. 2022) (siehe S. 2).

[2] Michael J. Heron, Vicki L. Hanson und Ian Ricketts. "Open Source and Accessibility: Advantages and Limitations". en. In: *Journal of Interaction Science* 1.1 (2013), S. 2. ISSN: 2194-0827. DOI: 10.1186/2194-0827-1-2. URL: http://journalofinteractionscience.springeropen.com/articles/10.1186/2194-0827-1-2 (besucht am 25. 05. 2022) (siehe S. 2, 18, 19, 28).

[3] GO FAIR. *GO-FAIR Website*. English. URL: https://www.go-fair.org/fair-principles/ (besucht am 26. 05. 2022) (siehe S. 2, 16, 33).

[4] Wilhelm Hasselbring u. a. "From FAIR research data toward FAIR and open research software". en. In: *it - Information Technology* 62.1 (Feb. 2020), S. 39–47. ISSN: 2196-7032, 1611-2776. DOI: 10.1515/itit-2019-0040. URL: https://www.degruyter.com/document/doi/10.1515/itit-2019-0040/html (besucht am 01. 06. 2022) (siehe S. 2).

[5] Stefan Pfenninger u. a. "The importance of open data and software: Is energy research lagging behind?" en. In: *Energy Policy* 101 (Feb. 2017), S. 211–215. ISSN: 03014215. DOI: 10.1016/j.enpol.2016.11.046. URL: https://linkinghub.elsevier.com/retrieve/pii/S0301421516306516 (besucht am 20. 08. 2022) (siehe S. 2).

[6] Scarlet Rahy und Julian M. Bass. "Managing non-functional requirements in agile software development". en. In: *IET Software* 16.1 (Feb. 2022), S. 60–72. ISSN: 1751-8806, 1751-8814. DOI: 10.1049/sfw2.12037. URL: https://onlinelibrary.wiley.com/doi/10.1049/sfw2.12037 (besucht am 01. 06. 2022) (siehe S. 2).

[7] Graham Lee u. a. "Barely sufficient practices in scientific computing". en. In: *Patterns* 2.2 (Feb. 2021), S. 100206. ISSN: 26663899. DOI: 10.1016/j.patter.2021.100206. URL: https://linkinghub.elsevier.com/retrieve/pii/S2666389921000167 (besucht am 01. 06. 2022) (siehe S. 2).

[8] Helmut Balzert und Peter Liggesmeyer. *Lehrbuch der Softwaretechnik. 2: Entwurf, Implementierung, Installation und Betrieb / Helmut Balzert. Unter Mitw. von Peter Liggesmeyer*. ger. 3. Auflage. Lehrbücher der Informatik. Heidelberg: Spektrum, Akademischer Verlag, 2011. ISBN: 978-3-8274-1706-0 (siehe S. 2).

[9] Stephen cass. *The Top Programming Languages 2019*. Sep. 2019. URL: https://spectrum.ieee.org/the-top-programming-languages-2019 (besucht am 01. 06. 2022) (siehe S. 3).

[10] *Leitfaden Definition*. 2022. URL: https://www.duden.de/node/88957/revision/1020944 (besucht am 05. 07. 2022) (siehe S. 4).

[11] Karl Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. Second. O'Reilly Media, Jan. 2017. URL: http://www.producingoss.com/ (besucht am 10. 08. 2022) (siehe S. 4).

[12] *DIN 66001:1983-12, Informationsverarbeitung; Sinnbilder und ihre Anwendung*. Techn. Ber. Beuth Verlag GmbH. DOI: 10.31030/1300312. URL: https://www.beuth.de/de/-/-/1080044 (besucht am 30. 07. 2022) (siehe S. 5).

[13] Gregor Büchel. *Praktische Informatik - Eine Einführung*. de. Wiesbaden: Vieweg+Teubner Verlag, 2012. ISBN: 978-3-8348-1874-4 978-3-8348-2283-3. DOI: 10.1007/978-3-8348-2283-3. URL: http://link.springer.com/10.1007/978-3-8348-2283-3 (besucht am 30. 07. 2022) (siehe S. 5).

[14] Chrklemm u. a. *chrklemm/SESMG: SESMG v0.0.4*. Sep. 2021. DOI: 10.5281/ZENODO.5412027. URL: https://zenodo.org/record/5412027 (besucht am 06. 07. 2022) (siehe S. 5, 39).

[15] Christian Klemm und Frauke Wiese. "Indicators for the optimization of sustainable urban energy systems based on energy system modeling". en. In: *Energy, Sustainability and Society* 12.1 (Dez. 2022), S. 3. ISSN: 2192-0567. DOI: 10.1186/s13705-021-00323-3. URL: https://energsustainsoc.biomedcentral.com/articles/10.1186/s13705-021-00323-3 (besucht am 06. 07. 2022) (siehe S. 5, 6).

[16] "R2Q – RessourcenPlan im Quartier". Deutsch. In: *BMBF und FONA* (). URL: https://ressourceneffiziente-stadtquartiere.de/?page_id=212&lang=de (besucht am 06. 07. 2022) (siehe S. 6, 49).

[17] Jakob Axelsson und Mats Skoglund. "Quality assurance in software ecosystems: A systematic literature mapping and research agenda". en. In: *Journal of Systems and Software* 114 (Apr. 2016), S. 69–81. ISSN: 01641212. DOI: 10.1016/j.jss.2015.12.020. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121215002861 (besucht am 08. 06. 2022) (siehe S. 9).

[18] Terhi Kilamo u. a. "From proprietary to open source—Growing an open source ecosystem". en. In: *Journal of Systems and Software* 85.7 (Juli 2012), S. 1467–1478. ISSN: 01641212. DOI: 10.1016/j.jss.2011.06.071. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121211001683 (besucht am 08. 06. 2022) (siehe S. 9, 10, 19, 20).

[19] Lothar Borrmann. "Zum Geleit: Software-Eco-Systeme". In: *Marktplätze im Umbruch*. Hrsg. von Claudia Linnhoff-Popien, Michael Zaddach und Andreas Grahl. Series Title: Xpert.press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, S. 627–629. ISBN: 978-3-662-43781-0 978-3-662-43782-7. DOI: 10.1007/978-3-662-43782-7_66. URL: http://link.springer.com/10.1007/978-3-662-43782-7_66 (besucht am 06. 07. 2022) (siehe S. 9).

[20] Kumiyo Nakakoji u. a. "Evolution patterns of open-source software systems and communities". en. In: *Proceedings of the international workshop on Principles of software evolution - IWPSE '02*. Orlando, Florida: ACM Press, 2002, S. 76. ISBN: 978-1-58113-545-9. DOI: 10.1145/512035.512055. URL: http://portal.acm.org/citation.cfm?doid=512035.512055 (besucht am 10. 06. 2022) (siehe S. 9).

[21] Peter A. Gloor. *Swarm creativity: competitive advantage through collaborative innovation networks*. Oxford ; New York: Oxford University Press, 2006. ISBN: 978-0-19-530412-1 (siehe S. 9).

[22] Alexander Schatten u. a. *Best Practice Software-Engineering: eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. ger. Heidelberg: Spektrum Akademischer Verl, 2010. ISBN: 978-3-8274-2486-0 (siehe S. 10, 16).

[23] Institute of Electrical and Electronics Engineers. *IEEE recommended practice for software requirements specifications: approved 25 June 1998, IEEE-SA Standards Board*. eng. New York, NY: IEEE, 1998. ISBN: 978-0-7381-0332-7 (siehe S. 10, 11).

[24] Andreas Prlić und James B. Procter. "Ten Simple Rules for the Open Development of Scientific Software". en. In: *PLoS Computational Biology* 8.12 (Dez. 2012), e1002802. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1002802. URL: https://dx.plos.org/10.1371/journal.pcbi.1002802 (besucht am 12. 07. 2022) (siehe S. 11, 12, 28–30).

[25] Markus List, Peter Ebert und Felipe Albrecht. "Ten Simple Rules for Developing Usable Software in Computational Biology". en. In: *PLOS Computational Biology* 13.1 (Jan. 2017). Hrsg. von Scott Markel, e1005265. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1005265. URL: https://dx.plos.org/10.1371/journal.pcbi.1005265 (besucht am 12. 07. 2022) (siehe S. 11, 28).

[26] Nazatul Nurlisa Zolkifli, Amir Ngah und Aziz Deraman. "Version Control System: A Review". en. In: *Procedia Computer Science* 135 (2018), S. 408–415. ISSN: 18770509. DOI: 10.1016/j.procs.2018.08.191. URL: https://linkinghub.elsevier.com/retrieve/pii/S1877050918314819 (besucht am 19. 07. 2022) (siehe S. 11–13).

[27] Morgan Taschuk und Greg Wilson. "Ten simple rules for making research software more robust". en. In: *PLOS Computational Biology* 13.4 (Apr. 2017), e1005412. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1005412. URL: https://dx.plos.org/10.1371/journal.pcbi.1005412 (besucht am 12. 07. 2022) (siehe S. 11, 16, 17, 22, 24, 25, 28, 29, 31, 43).

[28] Stefan Otte. "Version Control Systems". In: *Computer Systems and Telematics* (2009). URL: http://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2008-09_WS/S_19565_Proseminar_Technische_Informatik/otte09version.pdf (besucht am 19. 07. 2022) (siehe S. 12).

[29] C. Rodriguez-Bustos und J. Aponte. "How Distributed Version Control Systems impact open source software projects". In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. Zurich: IEEE, Juni 2012, S. 36–39. ISBN: 978-1-4673-1761-0 978-1-4673-1760-3. DOI: 10.1109/MSR.2012.6224297. URL: http://ieeexplore.ieee.org/document/6224297/ (besucht am 20. 07. 2022) (siehe S. 12).

[30] Yasset Perez-Riverol u. a. "Ten Simple Rules for Taking Advantage of Git and GitHub". en. In: *PLOS Computational Biology* 12.7 (Juli 2016). Hrsg. von Scott Markel, e1004947. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1004947. URL: https://dx.plos.org/10.1371/journal.pcbi.1004947 (besucht am 12. 07. 2022) (siehe S. 12, 13, 18, 26).

[31] *Google Trends: VCS-Vergleich*. 2022. URL: https://trends.google.de/trends/explore?date=2004-01-01%202022-07-01&geo=DE&q=Concurrent%20Version%20System,Apache%20Subversion,Git (besucht am 22. 07. 2022) (siehe S. 12, 13).

[32] Gede Artha Azriadi Prana u. a. "Categorizing the Content of GitHub README Files". In: (2018). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.1802.06997. URL: https://arxiv.org/abs/1802.06997 (besucht am 26. 05. 2022) (siehe S. 16, 17).

[33] Robert Crystal-Ornelas u. a. "A Guide to Using GitHub for Developing and Versioning Data Standards and Reporting Formats". en. In: *Earth and Space Science* 8.8 (Aug. 2021). ISSN: 2333-5084, 2333-5084. DOI: 10.1029/2021EA001797. URL: https://onlinelibrary.wiley.com/doi/10.1029/2021EA001797 (besucht am 26. 05. 2022) (siehe S. 16).

[34] Benjamin D. Lee. "Ten simple rules for documenting scientific software". en. In: *PLOS Computational Biology* 14.12 (Dez. 2018). Hrsg. von Scott Markel, e1006561. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1006561. URL: https://dx.plos.org/10.1371/journal.pcbi.1006561 (besucht am 12. 07. 2022) (siehe S. 17, 28–30).

[35] *Relative links and image paths in README files*. Documentation. 2022. URL: https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes#relative-links-and-image-paths-in-readme-files (besucht am 26. 05. 2022) (siehe S. 17).

[36] *README-File Prana et al.* URL: https://github.com/readmes/alt-blog.github.io/blob/master/README2.md (besucht am 26. 05. 2022) (siehe S. 17).

[37] Naoki Kobayakawa und Kenichi Yoshida. "How GitHub Contributing.md Contributes to Contributors". In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Turin: IEEE, Juli 2017, S. 694–696. ISBN: 978-1-5386-0367-3. DOI: 10.1109/COMPSAC.2017.139. URL: http://ieeexplore.ieee.org/document/8029680/ (besucht am 27. 07. 2022) (siehe S. 17, 18).

[38] *Starting an Open Source Project*. English. URL: https://opensource.guide/starting-a-project/ (besucht am 05. 08. 2022) (siehe S. 17).

[39] Omar Elazhary u. a. "Do as I Do, Not as I Say: Do Contribution Guidelines Match the GitHub Contribution Process?" In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Cleveland, OH, USA: IEEE, Sep. 2019, S. 286–290. ISBN: 978-1-72813-094-1. DOI: 10.1109/ICSME.2019.00043. URL: https://ieeexplore.ieee.org/document/8919187/ (besucht am 27. 07. 2022) (siehe S. 17, 18).

[40] The Apache Software Foundation. *ASF CONTRIBUTOR AGREEMENTS*. English. 2022. URL: https://www.apache.org/licenses/contributor-agreements.html#clas (besucht am 05. 08. 2022) (siehe S. 18).

[41] Inc. GitHub. *Github - Create Issue Templates*. English. Dokumentation. 2022. URL: https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/configuring-issue-templates-for-your-repository (besucht am 10. 10. 2022) (siehe S. 18).

[42] Inc. GitHub. *Github- Create PR template*. English. Dokumentation. 2022. URL: https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/creating-a-pull-request-template-for-your-repository (besucht am 10. 10. 2022) (siehe S. 18).

[43] O. Gotel und A. Finkelstein. "Extended requirements traceability: results of an industrial case study". In: *Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering*. Annapolis, MD, USA: IEEE Comput. Soc. Press, 1997, S. 169–178. ISBN: 978-0-8186-7740-3. DOI: 10.1109/ISRE.1997.566866. URL: http://ieeexplore.ieee.org/document/566866/ (besucht am 05. 08. 2022) (siehe S. 18).

[44] Renee Li u. a. "Code of Conduct Conversations in Open Source Software Projects on Github". en. In: *Proceedings of the ACM on Human-Computer Interaction* 5.CSCW1 (Apr. 2021), S. 1–31. ISSN: 2573-0142. DOI: 10.1145/3449093. URL: https://dl.acm.org/doi/10.1145/3449093 (besucht am 29. 07. 2022) (siehe S. 18, 19).

[45] Inc. GitHub. *Adding a code of conduct to your project - GitHub Docs*. Dokumentation. 2022. URL: https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/adding-a-code-of-conduct-to-your-project (besucht am 29. 07. 2022) (siehe S. 19).

[46] Inc. Free Software Foundation. *Copyleft. Was ist das?* Deutsch. März 2022. URL: https://www.gnu.org/licenses/copyleft.de.html (besucht am 05. 08. 2022) (siehe S. 19).

[47] Georgia M. Kapitsaki, Nikolaos D. Tselikas und Ioannis E. Foukarakis. "An insight into license tools for open source software systems". en. In: *Journal of Systems and Software* 102 (Apr. 2015), S. 72–87. ISSN: 01641212. DOI: 10.1016/j.jss.2014.12.050. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121214002945 (besucht am 02. 06. 2022) (siehe S. 19, 20).

[48] Inc. Free Software Foundation. *Freie Software. Was ist das?* Deutsch/English. Juli 2019. URL: https://www.gnu.org/philosophy/free-sw#n1 (besucht am 25. 07. 2022) (siehe S. 19).

[49] Maria Kechagia, Diomidis Spinellis und Stephanos Androutsellis-Theotokis. "Open Source Licensing Across Package Dependencies". In: *2010 14th Panhellenic Conference on Informatics*. Tripoli, Greece: IEEE, Sep. 2010, S. 5600285. ISBN: 978-1-4244-7838-5. DOI: 10.1109/PCI.2010.28. URL: http://ieeexplore.ieee.org/document/5600285/ (besucht am 02. 06. 2022) (siehe S. 20).

[50] Inc. Free Software Foundation. *Verschiedene Lizenzen und Kommentare*. Deutsch/English. Dez. 2018. URL: https://www.gnu.org/licenses/license-list.html#GPLCompatibleLicenses (besucht am 25. 07. 2022) (siehe S. 20, 21).

[51] Inc. Free Software Foundation. *Why you shouldn't use the Lesser GPL for your next library*. Jan. 2022. URL: https://www.gnu.org/licenses/why-not-lgpl.html.en (besucht am 02. 06. 2022) (siehe S. 20).

[52] Open Source Initiative and others. "Report of license proliferation committee". In: (2006). URL: https://opensource.org/proliferation-report (besucht am 25. 07. 2022) (siehe S. 20).

[53] Georgia M. Kapitsaki, Frederik Kramer und Nikolaos D. Tselikas. "Automating the license compatibility process in open source software with SPDX". en. In: *Journal of Systems and Software* 131 (Sep. 2017), S. 386–401. ISSN: 01641212. DOI: 10.1016/j.jss.2016.06.064. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121216300905 (besucht am 01. 06. 2022) (siehe S. 20).

[54] Opensource.org. *Apache-Lizenz, Version 2.0*. Jan. 2004. URL: https://opensource.org/licenses/Apache-2.0 (besucht am 02. 06. 2022) (siehe S. 20).

[55] Inc. Free Software Foundation. *How to Choose a License for Your Own Work*. Jan. 2022. URL: https://www.gnu.org/licenses/license-recommendations.html (besucht am 02. 06. 2022) (siehe S. 20).

[56] Opensource.org. *GNU General Public License*. URL: https://opensource.org/licenses/gpl-license (besucht am 02. 06. 2022) (siehe S. 20).

[57] *Open Source Initiative*. 2022. URL: https://opensource.org (besucht am 25. 07. 2022) (siehe S. 21).

[58] Mathias Meyer. "Continuous Integration and Its Tools". In: *IEEE Software* 31.3 (Mai 2014), S. 14–16. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2014.58. URL: https://ieeexplore.ieee.org/document/6802994/ (besucht am 29. 06. 2022) (siehe S. 21, 25).

[59] Inc. GitHub. *Github Actions*. English. Dokumentation. 2022. URL: https://github.com/features/actions (besucht am 10. 08. 2022) (siehe S. 21).

[60] Nick Coghlan und Donald Stufft. *Pep 440–version identification and dependency specification*. Techn. Ber. Tech. rep., Python Software Foundation, python Enhancement Proposal (PEP˜..., 2013. URL: https://github.com/python/peps/blob/main/pep-0440.txt (besucht am 15. 07. 2022) (siehe S. 22).

[61] Mahmoud Hashemi. *Calendar Versioning*. Juli 2019. URL: https://calver.org (besucht am 15. 07. 2022) (siehe S. 22).

[62] Kishori Sharan. *Java 13 Revealed: For Early Adoption and Migration*. en. Berkeley, CA: Apress, 2019. ISBN: 978-1-4842-5406-6 978-1-4842-5407-3. DOI: 10.1007/978-1-4842-5407-3. URL: http://link.springer.com/10.1007/978-1-4842-5407-3 (besucht am 15. 07. 2022) (siehe S. 22).

[63] Thorsten Lenk, Joachim W Schmidt und Florian Matthes. "Flexible Versionierung semistrukturierter Dokumente im Kontext von Autonomie und Kooperation". Diss. Hamburg: TUHH, 2001. URL: https://vmmatthes44.in.tum.de/file/1jgye9rxrdt1j/Sebis-Public-Website/-/2001/Lenk01/Lenk01.pdf (besucht am 15. 07. 2022) (siehe S. 22).

[64] Steven Raemaekers, Arie van Deursen und Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository". In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. Victoria, BC, Canada: IEEE, Sep. 2014, S. 215–224. ISBN: 978-1-4799-6148-1. DOI: 10.1109/SCAM.2014.30. URL: http://ieeexplore.ieee.org/document/6975655/ (besucht am 14. 07. 2022) (siehe S. 22, 23).

[65] Patrick Lam, Jens Dietrich und David J. Pearce. "Putting the semantics into semantic versioning". en. In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Virtual USA: ACM, Nov. 2020, S. 157–179. ISBN: 978-1-4503-8178-9. DOI: 10.1145/3426428.3426922. URL: https://dl.acm.org/doi/10.1145/3426428.3426922 (besucht am 14. 07. 2022) (siehe S. 23, 29–31).

[66] *Angular - Contribution Guideline*. Juni 2022. URL: https://github.com/angular/angular/blob/main/CONTRIBUTING.md#commit-message-header (besucht am 08. 08. 2022) (siehe S. 24).

[67] Haley Hunter-Zinck u. a. "Ten simple rules on writing clean and reliable open-source scientific software". en. In: *PLOS Computational Biology* 17.11 (Nov. 2021). Hrsg. von Scott Markel, e1009481. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1009481. URL: https://dx.plos.org/10.1371/journal.pcbi.1009481 (besucht am 12. 07. 2022) (siehe S. 24, 26, 27, 29).

[68] Moritz Lenz. *Python Continuous Integration and Delivery: A Concise Guide with Examples*. en. Berkeley, CA: Apress, 2019. ISBN: 978-1-4842-4280-3 978-1-4842-4281-0. DOI: 10.1007/978-1-4842-4281-0. URL: http://link.springer.com/10.1007/978-1-4842-4281-0 (besucht am 29. 06. 2022) (siehe S. 24–26).

[69] Guido van Rossum, Barry Warsaw und Nick Coghlan. *Style Guide for Python*. English. URL: https://github.com/python/peps/blob/main/pep-0008.txt (besucht am 29. 06. 2022) (siehe S. 24, 27).

[70] Mohd Ehmer Khan und Farmeena Khan. "Importance of software testing in software development life cycle". In: Bd. 11. International Journal of Computer Science Issues (IJCSI). 2014, S. 120. URL: https://www.ijcsi.org/papers/IJCSI-11-2-2-120-123.pdf (besucht am 29. 06. 2022) (siehe S. 24, 32).

[71] ISO. *ISO 26262: Road vehicles – Functional safety*. 2011. (Besucht am 29. 06. 2022) (siehe S. 25).

[72] Srinivas Nidhra. "Black Box and White Box Testing Techniques - A Literature Review". In: *International Journal of Embedded Systems and Applications* 2.2 (Juni 2012), S. 29–50. ISSN: 18395171. DOI: 10.5121/ijesa.2012.2204. URL: http://www.airccse.org/journal/ijesa/papers/2212ijesa04.pdf (besucht am 09. 08. 2022) (siehe S. 25).

[73] Michael Olan. "Unit testing: test early, test often". English. 2003. URL: https://dl.acm.org/doi/10.5555/948785.948830 (besucht am 07. 08. 2022) (siehe S. 25).

[74] Greg L. Turnquist. *Python testing cookbook: over 70 simple but incredibly effective recipes for taking control of automated testing using powerful Python testing tools*. eng. Quick answers to common problems. Birmingham: Packt Publ, 2011. ISBN: 978-1-84951-466-8 (siehe S. 25).

[75] Prachi Paigude u. a. "Software Integration Test Report Analysis Automation Using Python". In: *2021 Asian Conference on Innovation in Technology (ASIANCON)*. PUNE, India: IEEE, Aug. 2021, S. 1–6. ISBN: 978-1-72818-402-9. DOI: 10.1109/ASIANCON51346.2021.9544984. URL: https://ieeexplore.ieee.org/document/9544984/ (besucht am 29. 06. 2022) (siehe S. 25).

[76] Brian Okken. *Python testing with Pytest: simple, rapid, effective and scalable*. English. OCLC: 1322869892. 2022. ISBN: 978-1-68050-860-4 (siehe S. 26).

[77] Stefan Pfenninger u. a. "Opening the black box of energy modelling: Strategies and lessons learned". en. In: *Energy Strategy Reviews* 19 (Jan. 2018), S. 63–71. ISSN: 2211467X. DOI: 10.1016/j.esr.2017.12.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S2211467X17300809 (besucht am 23. 08. 2022) (siehe S. 26).

[78] Noah Fuhst. *PEPs-Python Enhancement Proposals*. Deutsch. URL: https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2019/pih-19-fuhst-pep-ausarbeitung.pdf (besucht am 29. 06. 2022) (siehe S. 27).

[79] J. Burton Browning und Marty Alchin. *Pro Python*. eng. 2. ed. Books for professionals by professionals. Berkeley, Calif.: Apress, 2014. ISBN: 978-1-4842-0334-7 (siehe S. 27).

[80] Emad Aghajani u. a. "Software documentation: the practitioners' perspective". en. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Juni 2020, S. 590–601. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380405. URL: https://dl.acm.org/doi/10.1145/3377811.3380405 (besucht am 31. 07. 2022) (siehe S. 28–30).

[81] openstack. *Deployment guides*. English. Documentation. Juli 2022. URL: https://docs.openstack.org/doc-contrib-guide/project-deploy-guide.html (besucht am 14. 08. 2022) (siehe S. 28).

[82] David Goodger und Guido van Rossum. *Docstring Conventions for Python*. English. URL: https://github.com/python/peps/blob/main/pep-0257.txt (besucht am 14. 08. 2022) (siehe S. 30).

[83] Moshe Zadka. *DevOps in Python: Infrastructure as Python*. en. Berkeley, CA: Apress, 2022. ISBN: 978-1-4842-7995-3 978-1-4842-7996-0. DOI: 10.1007/978-1-4842-7996-0. URL: https://link.springer.com/10.1007/978-1-4842-7996-0 (besucht am 11. 08. 2022) (siehe S. 31).

[84] Kristian Rother. *Pro Python Best Practices*. en. Berkeley, CA: Apress, 2017. ISBN: 978-1-4842-2240-9 978-1-4842-2241-6. DOI: 10.1007/978-1-4842-2241-6. URL: http://link.springer.com/10.1007/978-1-4842-2241-6 (besucht am 11. 08. 2022) (siehe S. 31).

[85] Meriem Belguidoum und Fabien Dagnat. "Dependency Management in Software Component Deployment". en. In: *Electronic Notes in Theoretical Computer Science* 182 (Juni 2007), S. 17–32. ISSN: 15710661. DOI: 10.1016/j.entcs.2006.09.029. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066107003830 (besucht am 12. 08. 2022) (siehe S. 31).

[86] Martin Fowler. *Refactoring: wie Sie das Design bestehender Software verbessern*. ger. Übers. von Jan Linxweiler und Knut Lorenzen. 2. Auflage. Frechen: mitp, 2020. ISBN: 978-3-95845-942-7 978-3-95845-941-0 (siehe S. 32–34, 51).

[87] Cory Kasper und Michael W. Godfrey. "Toward a Taxonomy of Clones in Source Code: A Case Study". In: (Besucht am 20. 06. 2022) (siehe S. 33, 35).

[88] Md. Monzur Morshed, Md. Arifur Rahman und Salah Uddin Ahmed. "A Literature Review of Code Clone Analysis to Improve Software Maintenance Process". In: (2012). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.1205.5615. URL: https://arxiv.org/abs/1205.5615 (besucht am 20. 06. 2022) (siehe S. 33).

[89] Sandro Schulze und Martin Kuhlemann. "Advanced Analysis for Code Clone Removal". In: *Proceedings des Workshops der GI-Fachgruppe Software Reengineering (SRE), erschienen in den GI Softwaretechnik-Trends 29 (2)*. 2009, S. 10 –12. (Besucht am 20. 06. 2022) (siehe S. 33).

[90] Chanchal K. Roy, James R. Cordy und Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". en. In: *Science of Computer Programming* 74.7 (Mai 2009), S. 470–495. ISSN: 01676423. DOI: 10.1016/j.scico.2009.02.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642309000367 (besucht am 22. 06. 2022) (siehe S. 33, 35).

[91] Peter Kokol, Marko Kokol und Sašo Zagoranski. "Code smells: A Synthetic Narrative Review". In: (2021). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2103.01088. URL: https://arxiv.org/abs/2103.01088 (besucht am 21. 06. 2022) (siehe S. 33).

[92] Ralph Steyer. *Programmierung in Python*. de. Wiesbaden: Springer Fachmedien Wiesbaden, 2018. ISBN: 978-3-658-20704-5 978-3-658-20705-2. DOI: 10.1007/978-3-658-20705-2. URL: http://link.springer.com/10.1007/978-3-658-20705-2 (besucht am 24. 08. 2022) (siehe S. 35).

[93] Peter Bulychev und Marius Minea. "Duplicate code detection using anti-unification". English. In: 2008. URL: http://clonedigger.sourceforge.net/duplicate_code_detection_bulychev_minea.pdf (besucht am 22. 06. 2022) (siehe S. 35).

[94] *SESMG Examples – Repository*. 2022. URL: https://github.com/chrklemm/SESMG_Examples (besucht am 06. 07. 2022) (siehe S. 39).

[95] *Volltextdokumentation des SESMG – RTD*. English. URL: https://readthedocs.org/projects/spreadsheet-energy-system-model-generator/ (besucht am 06. 07. 2022) (siehe S. 39).

[96] *Sphinx - automated Documentation*. 2022. URL: https://github.com/sphinx-doc/sphinx (besucht am 06. 07. 2022) (siehe S. 39).

[97] Christian Klemm und Peter Vennemann. "Modeling and optimization of multi-energy systems in mixed-use districts: A review of existing methods and approaches". en. In: *Renewable and Sustainable Energy Reviews* 135 (Jan. 2021), S. 110206. ISSN: 13640321. DOI: 10.1016/j.rser.2020.110206. URL: https://linkinghub.elsevier.com/retrieve/pii/S1364032120304950 (besucht am 14. 08. 2022) (siehe S. 40).

[98] shields.io. *Github Badges*. URL: https://github.com/badges/shields (besucht am 20. 08. 2022) (siehe S. 50).

[99] Inc. GitHub. *GitHub glossary*. Documentation. 2022. URL: https://docs.github.com/en/get-started/quickstart/github-glossary (besucht am 15. 08. 2022) (siehe S. 50, 51).

[100] Python Software Foundation. *Glossary*. Documentation. 2022. URL: https://docs.python.org/3/glossary.html (besucht am 15. 08. 2022) (siehe S. 50).

[101] Henry Eickhoff. *Quellcode: Begriffserklärung, Aufbau und Erstellung*. Mai 2022. URL: https://blog.hubspot.de/website/quellcode (besucht am 15. 08. 2022) (siehe S. 50).

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit „Bereitstellung von Open Source Software in der Energiewirtschaft – Ein Leitfaden“ selbstständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Literaturquellen sind im Literaturverzeichnis vollständig zitiert.

Gregor Becker, 26. August 2022