Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Die Veröffentlichung einer Open-Source-Software aus dem universitären Kontext am Beispiel von RENEW: Dokumentation und Analyse des Releaseprozesses und Betrachtung von möglichen Anpassungen

vorgelegt von

Relana Streckenbach

Fakultät MIN-Fakultät

Fachbereich Informatik

Arbeitsbereich: Algorithmen, Randomisierung, Theorie

Studiengang: Bachelor Informatik

Matrikelnummer: 7396590

Erstgutachter: Dr. Daniel Moldt

Zweitgutachter: David Mosteller

Zusammenfassung. Die Veröffentlichung von Software ist ein wichtiger Schritt im Entwicklungsprozess, da sie das Produkt für Nutzende außerhalb des Entwicklungsteams verfügbar macht. Im Allgemeinen ist die Freigabe von Software keine triviale Aufgabe, da sie bestenfalls umfassende Qualitätssicherungsmaßnahmen, eine gut durchdachte Dokumentation und Versionierung, ein robustes Bauen und Packaging der Software sowie geeignete Vertriebskanäle umfasst. Diese Aufgabe ist besonders herausfordernd in Umgebungen, in denen das Entwicklungsteam Fluktuationen unterliegt und der Prozess weder gut dokumentiert noch automatisiert ist.

Die seit langem bestehende Open-Source-Software RENEW wird an der Universität Hamburg entwickelt und gepflegt. Der Releaseprozess wird größtenteils manuell durchgeführt und die notwendigen Schritte sind unzureichend dokumentiert. Aufgrund des universitären Umfelds variieren die ständig wechselnden Entwickelnden in ihren Fähigkeiten und Erfahrungen bzgl. der Entwicklung der Software. Aufgrund dieser Gegebenheiten ist die Veröffentlichung von RENEW-Versionen ein mühsamer Prozess und die Häufigkeit der Releases ist geringer als sie womöglich sein könnte.

Die Releases von RENEW 2.6, 4.0 und 4.1 wurden von dieser Arbeit begleitet, wobei die notwendigen Kenntnisse und Aufgaben identifiziert wurden, die im Rahmen des Releaseprozesses durchzuführen sind. Ziel dieser Arbeit ist es, den aktuellen Releaseprozess von RENEW gründlich und strukturiert zu dokumentieren, um einen umfassenden Überblick bereitzustellen, welcher von RENEW-Entwickelnden für zukünftige Releases und als Grundlage für die Automatisierung genutzt werden kann. Darüber hinaus wird eine Analyse des gesamten Prozesses angestrebt, innerhalb derer dessen aktuelle Herausforderungen und Fallstricke identifiziert werden. Für die identifizierten Probleme werden Verbesserungs- und Automatisierungsvorschläge auf der Grundlage von in der Literatur beschriebenen Best Practices gemacht.

Abstract. The release of software is an important step within its development process since it makes the product available to users outside of the development team. In general, releasing software is no trivial feat as it ideally encompasses comprehensive quality assurance measures, well thought out documentation and versioning, robust building and packaging of the product, as well as appropriate distribution channels. The task is particularly challenging in environments in which the development team is subject to fluctuation and the process is neither well documented nor automated.

The longstanding open source software RENEW is being developed and maintained at the University of Hamburg. Its release process is mostly done manually and the necessary steps are insufficiently documented. Due to the university setting the ever-changing developers vary in skill and experience within the development of the software. Owing to these given conditions releasing versions of RENEW is an arduous process and the frequency of releases is lower than it probably could be.

The releases of RENEW 2.6, 4.0, and 4.1 were accompanied by this work, whereby the necessary acquirements and tasks to be executed within the release process where identified. This thesis aims at thoroughly documenting the current release process of RENEW in a structured manner in order to provide a comprehensive overview which can be used by RENEW developers for future releases and as a basis for automation. It furthermore strives to present an analysis of the whole process within which its current challenges and pitfalls are identified. For the identified issues suggestions for improvement and automatisation are provided based on best practices found in literature.

Inhaltsverzeichnis

Anhänge

Kapitel 1

Einleitung

In diesem Kapitel wird zunächst die Motivation für die Erstellung dieser Arbeit beschrieben, bevor die Fragestellung und Zielsetzung der Ausarbeitung dargelegt werden. Zuletzt wird der Aufbau der Arbeit kapitelweise wiedergegeben.

1.1 Motivation

Vertrauen in die Qualität von Software ist wichtig. Veröffentlichte Versionen sollten robust und gut getestet sein (Sommerville 2018, Kap.8, 10). Zusätzlich ist es erstrebenswert Neuerungen möglichst zügig zu veröffentlichen, um Releaseprozesse risikoärmer zu gestalten und Erweiterungen der Software der jeweiligen Zielgruppe schnell zugänglich zu machen (Fowler 2013; Humble 2017a).

Der Veröffentlichungsprozess der Open-Source-Software RENEW, welche innerhalb universitärer Projekte und Abschlussarbeiten weiterentwickelt wird, ist zur Zeit aufwändig.

Das liegt zum einen daran, dass das Wissen über die notwendigen Arbeitsschritte und Abläufe nicht gebündelt und einfach abrufbar zur Verfügung steht. Zum anderen müssen die meisten für eine Veröffentlichung auszuführenden Schritte händisch durchgeführt werden.

Außerdem sind diese manuell ausgeführten, sich wiederholenden Tätigkeiten fehleranfällig, da die ständig wechselnden an der Entwicklung beteiligten Personen sich das Wissen über die Abläufe erst aneignen müssen und dabei Arbeitsschritte leicht außer Acht gelassen werden können.

Der hohe Aufwand manifestiert sich unter anderem darin, dass die für die Bereitstellung neuer Funktionen benötigte Zeit sehr lang ist bzw. neue Releases nur selten gemacht werden. Eine Anpassung des Vorgehens bzw. die Automatisierung der Abläufe wären erstrebenswert um eine stets aktuelle und stabile Version der Software bereitstellen zu können.

Aus diesen Gründen ist es sinnvoll und wichtig, den gesamten Releaseprozess am Beispiel der Veröffentlichung der jüngsten Versionen von RENEW zu dokumentieren und zu untersuchen.

1.2 Fragestellung und Zielsetzung

Diese Arbeit beschäftigt sich mit der Frage, wie sich der Releaseprozess der Open-Source-Software RENEW gestaltet, welche Akteure und Techniken daran beteiligt sind und wo u.U.

Herausforderungen insbesondere im Hinblick auf die Durchführung und die Automatisierung von Veröffentlichungen bestehen. Ferner werden mögliche Optionen diese Schwachstellen zu beheben betrachtet.

Die Dokumentation soll die Veröffentlichung zukünftiger Versionen von RENEW vereinfachen, indem das bestehende Wissen gesammelt und gebündelt aufbereitet wird. Die Durchführung des gesamten Releaseprozesses, also die Veröffentlichung von Versionen, die Aggregation des Wissens und die Verstetigung dieses Wissens sind die Kernaspekte dieser Arbeit.

Bei der Beschreibung von Herausforderungen innerhalb der Abläufe und dem Aufzeigen von möglichen Lösungen für diese soll Wert darauf gelegt werden, dass die Entwicklung ohne Qualitätsverlust an Zuverlässigkeit und Schnelligkeit gewinnt.

1.3 Aufbau der Arbeit

Zuerst legt das Grundlagenkapitel zentrale Begriffe und Konzepte dar.

Anschließend beschreibt die Anforderungsanalyse, welche Anforderungen im Hinblick auf Dokumentation, Codequalität und Automatisierung bestehen.

Im Kapitel „Organisation und Struktur der Veröffentlichungspraxis“ werden die Organisation des Releaseprozesses, die Versionierung und der Aufbau von Releases, Branching-Konventionen und Qualitätssicherung, sowie Buildmanagement und CI/CD-Pipelines innerhalb der Entwicklung von RENEW dargelegt, woraufhin im Kapitel „Erarbeitetes Vorgehen zur Durchführung eines Release“ eine Dokumentation davon geliefert wird, wie neue Versionen von RENEW für die Veröffentlichung vorbereitet und auf der Webseite veröffentlicht werden.

Danach wird die Veröffentlichungspraxis im Kapitel „Fragestellungen im Rahmen der Veröffentlichung“ bzgl. etwaiger Herausforderungen analysiert und es werden Vorschläge gemacht, wie diesen begegnet werden kann.

Es folgt eine Evaluation der vorliegenden Arbeit bezüglich der zu Beginn festgelegten Anforderungen.

Abschließend enthält der Schluss eine Zusammenfassung der Arbeit und einen Ausblick auf mögliche Themen und Fragen, die sich aus ihr ergeben.

Kapitel 2

Grundlagen

In diesem Kapitel werden die allgemeinen Grundlagen dargelegt, welche für diese Arbeit wichtig sind. Die Reihenfolge wurde so gewählt, dass möglichst wenig Vorwärtsreferenzen gemacht werden müssen. In späteren Kapiteln wird dediziert auf RENEW Bezug genommen.

Zuerst wird das Open-Source-Softwareprojekt RENEW vorgestellt, dessen Releaseprozess Untersuchungsgegenstand dieser Arbeit ist.

Im Abschnitt „Universitärer Kontext“ wird das Entwicklungsumfeld von RENEW kurz erläutert.

Es folgt eine Definition des Open-Source-Begriffs, wobei auch auf Besonderheiten der Softwareentwicklung in Open-Source-Projekten eingegangen wird.

Danach wird das Projektmanagement-Framework Scrum, welches bei der Entwicklung von RENEW eingesetzt wird, vorgestellt.

Anschließend werden Grundbegriffe der Versionsverwaltung mit Git dargelegt; Git wird in der Entwicklung von RENEW für die Versionsverwaltung genutzt.

Der Abschnitt „Releaseprozesse“ beschäftigt sich mit der Veröffentlichung von Software im Allgemeinen bzw. Open-Source-Software im Speziellen.

Am Ende werden die Konzepte von Continuous Integration und Continuous Delivery bzw. Deployment erläutert, bevor zuletzt die innerhalb von RENEWs Entwicklung verwendete Anwendung GitLab und insbesondere das Werkzeug GitLab CI/CD vorgestellt werden.

2.1 RENEW

RENEW[1] ist ein Softwareprojekt der ART-Gruppe[2] des Fachbereichs Informatik an der Universität Hamburg. Seit seiner ursprünglichen Entstehung vor über 25 Jahren[3] wird RENEW als Lehrprojekt desselben Arbeitsbereichs im Rahmen von Projektmodulen und Abschlussarbeiten gewartet und weiterentwickelt.

Da sie quelloffen ist, ist die Software sowohl für Studierende und Abschlussarbeitende als auch für externe interessierte Personen erweiterbar. Dieser Umstand bedeutet, dass es viele Kollaborateure gibt, die an RENEWs Entwicklung arbeiten.

RENEW wird innerhalb von Lehrmodulen wie „Modellierung und Analyse komplexer Systeme“[4] von Studierenden für die Bearbeitung von Übungsaufgaben verwendet.

Die Software ist in der Programmiersprache Java geschrieben.

[1] **Re**ference **Ne**t **W**orkshop, http://www.renew.de/

[2] Arbeitsbereich für **A**lgorithmen, **R**andomisierung und **T**heorie

[3] RENEW wurde am 05.03.1999 in der Version 1.0 veröffentlicht (Kummer und Wienberg 1999).

[4] https://www.inf.uni-hamburg.de/inst/ab/art/teaching/ws2023/vl-maks.html

Seit der ersten Version der Software können Referenz- und Petrinetze mit RENEW modelliert und simuliert werden (Kummer und Wienberg 1999).

Durch diverse Weiterentwicklungen ist RENEW zu einer erweiterbaren Petrinetz-IDE[5] geworden, mit der sich unterschiedliche Netzarten modellieren, ausführen und analysieren lassen (Cabac u. a. 2016; Moldt u. a. 2023b). Beispiele für Netze, die verwendet werden können, sind: Platz-/Transitions-Netze, Workflownetze und Referenznetze mit Java-Inskriptionen und der Möglichkeit der Modellierung von Netzen in Netzen (Clasen 2023).

Um RENEW auf einem aktuellen Stand zu halten und durch neue technische Anforderungen war es nötig, die Petrinetz-IDE nicht nur um Funktionalitäten zu erweitern, sondern z.B. auch die Architektur der Software, ihr Build-Management und ihr Continuous Integration(CI)-Pipeline-Tool zu aktualisieren.

Die Softwarearchitektur ist durch Plugins charakterisiert, diese wurden unter Verwendung des Java Platform Module Systems (JPMS) in Module umgewandelt (Clasen u. a. 2022; Daschkewitsch 2019; Janneck 2021).

Gleichzeitig zu der Modularisierung wurde das Build-Management-Automatisierungs-Tool gewechselt und zwar von Apache Ant zu Gradle (Clasen u. a. 2022).

Für die CI-Pipeline wird bei der Entwicklung von RENEW inzwischen GitLab CI/CD genutzt und nicht mehr Jenkins (Feldmann 2019).

2.2 Universitärer Kontext

RENEW ist eine Software, die im Rahmen universitärer Projekte und Abschlussarbeiten entwickelt wird. Aus diesem Grund gibt es wenig Permanenz im Entwicklungsteam, der Grad der fachlichen Kompetenzen der Entwickelnden variiert sehr und es stehen weniger Ressourcen als in einer Gruppe von Vollzeitentwickelnden in der Praxis zur Verfügung. Das Entwicklungsumfeld ist also charakterisiert durch hohe Fluktuation bei den an der Entwicklung teilnehmenden Personen, schlecht einzuschätzende Erfahrungsstände und wenig gleichzeitige, fortlaufende Arbeitszeiten.

2.3 Open Source

Wenn Software veröffentlicht wird, geschieht dies meist mit für sie geltenden Lizenzen. Eine Unterkategorie aller möglichen Lizenzen hierbei wird *Open Source* (dt.: quelloffen) genannt. Open Source Software (OSS) ist vordergründig dadurch charakterisiert, dass zusätzlich zum ausführbaren Programm ihr Quelltext veröffentlicht wird, sodass dieser der Öffentlichkeit zugänglich ist. Dieser Quelltext darf von den Nutzenden angeschaut, verändert und weiterverteilt werden.

Im Folgenden wird die genaue Definition für den Begriff Open Source zitiert. Des Weiteren wird der typische Entwicklungszyklus von OSS dargelegt, wie er von Feller und Fitzgerald definiert wurden.

2.3.1 Open-Source-Definition

Die Open Source Initiative (OSI) hat eine Open-Source-Definition (OSD) entworfen, die auf ihrer Webseite veröffentlicht ist (The Open Source Initiative 2023b). Die OSD wird von der

[5] **I**ntegrated **D**evelopment **E**nvironment

OSI verwaltet und wird standardmäßig als Definition von OSS verwendet (Feller und Fitzgerald 2002).

Diese Definition enthält zehn Kriterien, die in den Distributionsbedingungen von OSS gegeben sein müssen, damit sie als Open Source gilt (The Open Source Initiative 2023b). Zusammengefasst muss für OSS der Quelltext zugänglich gemacht werden und es dürfen wenige bis keine Beschränkungen für die Nutzung, Veränderung und Weiterverteilung bestehen. Eine Übersetzung der zehn Kriterien findet sich in Anhang A.

2.3.2 Open-Source-Softwareentwicklungs-Lebenszyklus

Den Entwicklungs-Lebenszyklus von OSS charakterisieren Feller und Fitzgerald anders als den traditionellen Softwareentwicklungs-Lebenszyklus (SDLC[6]), welchen sie mit den Phasen *Planung*, *Analyse*, *Design* und *Implementation* wiedergeben (Feller und Fitzgerald 2002, S.101). Andere Definitionen der traditionellen Softwareentwicklungsphasen weichen ab. Sommerville bspw. beschreibt das Wasserfallmodell, welches einen traditionellen SDLC beschreibt, mit den Phasen *Analyse und Definition der Anforderungen*, *System- und Softwareentwurf*, *Implementierung und Modultests*, *Integration und Systemtest*, *Betrieb und Wartung* (Sommerville 2018, S.57).

Nichtsdestotrotz lassen sich die Beobachtungen von Feller und Fitzgerald zum Unterschied zwischen traditioneller und Open-Source-Softwareentwicklung gut nachvollziehen. Die Phasen des Testens und der Wartung scheinen sie als Teil der Implementationsphase anzusehen. Ihre Beobachtungen lauten folgendermaßen: Der:die ursprüngliche Projektgründer:in nähme oft bereits die Planung, die Analyse und das Design vor. Diese Vorarbeiten sollten wohlüberlegt sein, da sie das OSS-Projekt prägen (Feller und Fitzgerald 2002, S.101f). OSS sollte gemäß den Autoren modular sein, damit später eine verteilte Entwicklung durch verschiedene mitwirkende Personen möglich ist (Feller und Fitzgerald 2002, S.101f). Außerdem sollten Designentscheidungen aus etablierten Entwurfsmustern folgen (Feller und Fitzgerald 2002, S.101f).

Der SDLC von OSS sei vornehmlich in der Implementationsphase anzusiedeln und unter Bezugnahme auf Jørgensens Arbeit, in der er den Lebenszyklus für Änderungen am FreeBSD Projekt beschreibt (Jørgensen 2001), wird dieser wie folgt charakterisiert:

code

Um neue Funktionalitäten oder Verbesserungen zum Projekt beizutragen, muss zunächst der benötigte Code geschrieben werden. Feller und Fitzgerald merken außerdem an, dass das Wissen darum, dass erfahrenere und respektierte Fachkolleg:innen den Code prüfen werden, die Entwickelnden dazu animiere, gut geschriebenen Code zu produzieren (Feller und Fitzgerald 2002, S.102ff).

review

Die entwickelnden Personen sollten, noch bevor sie ihren Code mergen (s. 2.5.4 Merge und Merge-Konflikt), häufig Feedback einfordern und bekommen. Dies sei essentiell für den Entwicklungsprozess, allerdings beschreiben die Autoren auch, dass dieses Feedback nicht leicht zu bekommen sei, wenn die Person noch nicht lange an der Entwicklung beteiligt ist oder der Code eine hohe Komplexität aufweist(Feller und Fitzgerald 2002, S.102ff).

[6] **S**oftware **D**evelopment **L**ife **C**ycle

pre-commit test

Das OSS-Entwicklungsmodell wird als „vulnerabel“ bezeichnet, weswegen es entscheidend sei, dass der geschriebene Code gründlich getestet wird. Dies ist notwendig, um sicher zu gehen, dass die Änderungen später nicht den Build kaputt machen (Feller und Fitzgerald 2002, S.102ff).

development release

Falls ein neu entwickeltes Modul von einem:r Committer:in (s. 2.5.2 Commit) als fertig angesehen wird, kann es in den *development release* (dt. etwa: Entwicklungsveröffentlichung) integriert werden (Feller und Fitzgerald 2002, S.102ff).

parallel debugging

Dieser Prozess wird als global und parallel beschrieben. Wenn Fehler gefunden werden, können sie entweder sofort behoben werden oder es können Problemberichte erstellt werden. Die Anzahl der potenziellen Debugger kann bei der OSS-Entwicklung hoch sein (Feller und Fitzgerald 2002, S.102ff). Jørgensen beschreibt allerdings das Problem, dass viele Fehlerberichte in dieser Phase nicht sonderlich hilfreich sind, da offensichtliche Probleme zwar sofort behoben werden, aber subtilere nicht gefunden werden (Jørgensen 2001).

production release

Änderungen, die alle vorherigen Phasen erfolgreich durchlaufen haben, werden schließlich Teil der nächsten Veröffentlichung. Dafür werden sie in den stabilen Produktionsbranch (engl.: production branch) (s. 2.5.3 Branch) integriert. Wie genau dieser Prozess organisiert ist, variiert von Projekt zu Projekt (Feller und Fitzgerald 2002, S.102ff).

2.4 Scrum

Scrum ist ein Projektmanagement-Framework, der besonders in der agilen Softwareentwicklung zum Einsatz kommt. Dieser Framework soll leichtgewichtig sein und helfen, Nutzen durch adaptive Lösungen für komplexe Probleme zu generieren (Schwaber und Sutherland 2020).

Da Scrum bzw. die Abwandlung *Scrum @ Scale* auch in den Lehrprojekten, in welchen RENEW weiterentwickelt wird, verwendet wird, sollen hier einige Grundbegriffe bezüglich der Rollen, Abläufe (Events) und Artefakte kurz erläutert werden[7]. Eine grafische Repräsentation der Implementation des Frameworks findet sich in Abbildung 2.1.

2.4.1 Rollen

Ein Scrum Team besteht aus Entwickelnden (engl.: Developer), einer Person als Scrum Master (SM) und einer Person als Product Owner (PO).

Developer sind die Personen, die determiniert sind, in jedem Sprint irgendeinen Aspekt eines nutzbaren Inkrements zu erstellen. Sie erstellen außerdem zusammen den Sprint Backlog (Schwaber und Sutherland 2020).

[7]Die Definition des Scrum-Frameworks in “The 2020 Scrum Guide” von Schwaber und Sutherland ist bereits recht kurz gehalten und sollte für eine Anwendung von Scrum komplett gelesen und umgesetzt werden. Hier werden nur die für diese Arbeit als wichtig erachteten Aspekte stark verkürzt wiedergegeben.

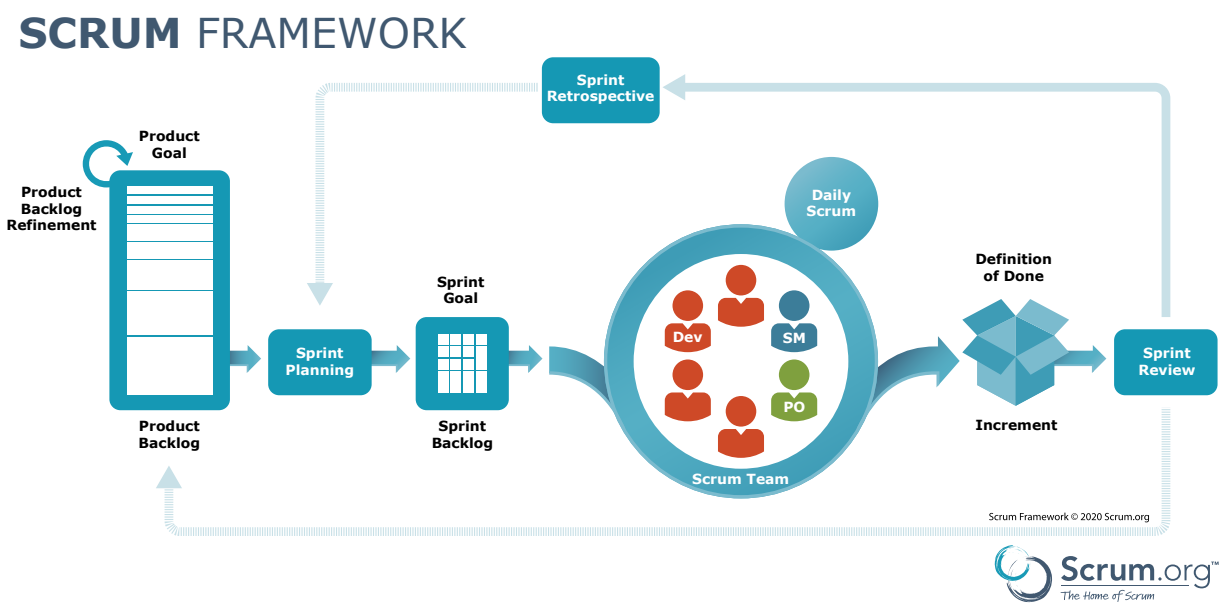


Abbildung 2.1: Scrum Framework (Scrum.org 2020)

SM sind verantwortlich dafür, alle Voraussetzungen zu schaffen, die nötig sind, damit Scrum nach den Regeln des Scrum Guides stattfinden kann. Die Person in dieser Rolle trägt außerdem die Verantwortlichkeit für die Effektivität des Scrum Teams; dazu gehört das Beseitigen von Hindernissen in der Entwicklung des Teams (Schwaber und Sutherland 2020).

PO sind dafür verantwortlich, den Wert des Produkts zu maximieren, welches aus der Arbeit des Scrum Teams resultiert. Die Person in dieser Rolle verantwortet das allgemeine Product Goal und das Product Backlog (Schwaber und Sutherland 2020).

Bei der Verwendung von Scrum @ Scale, bei dem es mehrere Scrum Teams gibt, existieren zusätzlich die Rollen Chief Product Owner (CPO) und Scrum of Scrums Master (SoSM). Der bzw. die CPO koordiniert das Product Backlog über die unterschiedlichen Scrum Teams hinweg. Der bzw. die SoSM ist dafür verantwortlich, die Koordination und Kollaboration zwischen den Scrum Teams sicherzustellen (Anonymous Author(s) 2024).

Außerdem existiert die Rolle Chief of Chief Product Owner (CCPO). Diese Rolle wird vom Modulverantwortlichen der universitären Lehrprojekte ausgefüllt.

2.4.2 Ereignisse

Die Entwicklung mit Scrum ist durch *Sprints* strukturiert. Sprints haben i.d.R. eine Länge von zwei oder vier Wochen, wobei der nächste Sprint sich direkt an das Ende des vorhergegangenen Sprints anschließt. Innerhalb eines Sprints finden neben der eigentlichen Arbeit an den Inkrementen unterschiedliche Events statt; nämlich das Sprint Planning, das Daily Scrum, der Sprint Review und die Sprint Retrospective. Die Events sollen Transparenz herstellen und die Möglichkeit bieten, Scrum Artefakte zu begutachten und anzupassen (Schwaber und Sutherland 2020).

Jeder Sprint beginnt mit einem *Sprint Planning*. In diesem wird das Ziel des aktuellen Sprints (Sprint Goal) definiert, welches zeigen soll, warum dieser Sprint wertschöpfend sein wird. Aus dem Product Backlog werden Punkte gewählt, die in diesem Sprint abgearbeitet werden sollen. Außerdem wird auch überlegt, wie diese Punkte zielführend bearbeitet werden. Das Sprint Goal, die gewählten Punkte aus dem Product Backlog und der Plan für ihre Bearbeitung ergeben zusammen den Sprint Backlog (Schwaber und Sutherland 2020).

Das *Daily Scrum* ist ein 15-minütiges Event, das an jedem Arbeitstag innerhalb eines Sprints vom Scrum Team abgehalten wird. In jedem Daily Scrum wird der Fortschritt und das geplante weitere Vorgehen der Teammitglieder kommuniziert. Durch Daily Scrums wird die Kommunikation verbessert, Hindernisse werden identifiziert, eine schnelle Entscheidungsfindung wird gefördert und die Notwendigkeit anderer Meetings wird im besten Fall eliminiert (Schwaber und Sutherland 2020).

Der *Sprint Review* ist das vorletzte Event eines Sprints. Hier werden die Ergebnisse des Sprints vom Scrum Team an die Stakeholder kommuniziert und gemeinsam der Fortschritt im Hinblick auf das Product Goal diskutiert. Außerdem sind Anpassungen am Product Backlog im Sprint Review möglich (Schwaber und Sutherland 2020).

In der *Sprint Retrospective* liegt der Fokus auf Qualitäts- und Effektivitätssteigerung. Das Scrum Team diskutiert gemeinsam, was im Sprint gut gelaufen ist, welche Probleme aufgetreten sind und wie diese Probleme gelöst bzw. nicht gelöst wurden. Das Scrum Team identifiziert anschließend die hilfreichsten Änderungen zur Verbesserung seiner Effektivität, um diese im nächsten Sprint umzusetzen (Schwaber und Sutherland 2020).

2.4.3 Artefakte

Laut Schwaber und Sutherland repräsentieren Scrum-Artefakte Arbeit oder Wert. Sie sollen die Transparenz von Schlüsselinformationen maximieren. Die in “The 2020 Scrum Guide” definierten Scrum-Artefakte sind: das Product Backlog, das Sprint Backlog und das Inkrement. Jedes Artefakt enthält ein Commitment (dt.: Verpflichtung, Verbindlichkeit), welches die Transparenz und den Fokus des Artefakts erhöhen soll. Am Commitment soll der jeweilige Fortschritt messbar sein (Schwaber und Sutherland 2020).

Das *Product Backlog* ist eine sich entwickelnde, geordnete Liste von Elementen, welche benötigt werden, um das Produkt zu verbessern. Alle Arbeiten des Scrum Teams, leiten sich aus dem Product Backlog ab. Sein Commitment ist das *Product Goal*, welches einen zukünftigen Zustand des Produkts definiert und das Langzeitziel des Scrum Teams darstellt. Der Rest des Product Backlogs dient dazu zu definieren, was das Product Goal erfüllen wird (Schwaber und Sutherland 2020).

Das *Sprint Backlog* besteht aus den drei im Sprint Planning erarbeiteten Elementen: dem Sprint Goal, den gewählten Punkten aus dem Product Backlog und dem Plan für ihre Bearbeitung. Schwaber und Sutherland bezeichnen diese drei Elemente als *why*, *what* und *how*. Das *Sprint Goal* ist das Commitment des Sprint Backlogs. Es soll für das Scrum Team Kohärenz und Fokus schaffen (Schwaber und Sutherland 2020).

Ein *Inkrement* ist ein konkreter Schritt in Richtung des Product Goals. Neue Inkremente müssen einen zusätzlichen Wert zu vorherigen schaffen, mit allen anderen Inkrementen zusammen funktionieren und benutzbar sein. Innerhalb eines Sprints können mehrere Inkremente fertig gestellt werden. Sie müssen hierfür die *Definition of Done* erfüllen, welche ihr Commitment ist. Die Definition of Done ist die formale Beschreibung des Zustands eines Inkrements, welches die für das Produkt benötigten Qualitätsanforderungen erfüllt (Schwaber und Sutherland 2020).

2.5 Versionsverwaltung mit Git

Bei der Entwicklung von Programmcode wird inzwischen meistens ein Versionsverwaltungssystem (VCS[8]) genutzt. Der Einsatz eines VCS bedeutet, dass vorherige Zustände von Dateien einfacher wiederhergestellt werden können (Chacon und Straub 2014, Kap.1). Git ist

[8] engl.: **V**ersion **C**ontrol **S**ystem

ein solches VCS bzw. verteiltes Versionsverwaltungssystem (DVCS[9]) und Softwareprojekte, welche Git als VCS nutzen, können z.B. mit der Webanwendung GitLab verwaltet werden.

Preißel und Stachmann nennen folgende Aspekte als Vorteile, die sich durch die Nutzung eines solchen dezentralen Systems ergeben (Preißel und Stachmann 2019, S. 3):

- Hohe Performance
- Effiziente Arbeitsweisen
- Offline-Fähigkeit
- Flexibilität der Entwicklungsprozesse
- Backup
- Wartbarkeit

All diese Eigenschaften sind von großem Vorteil bei der Softwareentwicklung im universitären Kontext.

Für das Arbeiten mit Git ist das Verständnis bestimmter Begriffe und Konzepte wichtig. Einige zentrale Grundbegriffe werden hier kurz erläutert.

2.5.1 Git-Repository und Working-Directory

Wenn Git für die Versionsverwaltung eines Projekts genutzt werden soll, muss zunächst ein *Git-Repository* in dem Ordner angelegt werden, in dem sich das Projekt befindet. Dies wird mit dem Befehl `git init` erzielt (Chacon und Straub 2014, Kap.2). Ab diesem Zeitpunkt können die Zustände der Dateien im Projektordner als Snapshots (dt.: Momentaufnahmen) von Git erfasst werden (Chacon und Straub 2014, Kap.1).

Es ist auch möglich ein bereits existierendes Repository zu klonen, dafür wird der Befehl `git clone [url]` verwendet (Chacon und Straub 2014, Kap.2). Nach der Ausführung des Befehls befindet sich eine Kopie des geklonten Ordners inklusive des Git-Repositories auf der Maschine, von der aus geklont wurde.

Den Projektordner bzw. die geklonte Kopie nennt man *Working-Directory*.

2.5.2 Commit

Neue Dateien in einem Working-Directory sind zunächst *untracked*, was bedeutet, dass die Dateien noch nicht in einem vorherigen Snapshot von Git waren. Um sie von Git tracken zu lassen, *staged* man die entsprechenden Dateien zunächst mit dem Befehl `git add`. Damit sie im nächsten Snapshot gespeichert werden, werden sie anschließend mit dem Befehl `git commit` festgehalten (Chacon und Straub 2014, Kap.2).

Mit `git add` und `git commit` werden auch Änderungen an Dateien festgehalten, die bereits *tracked* sind, also schon in einem vorherigen Snapshot waren (Chacon und Straub 2014, Kap.2).

Der festgehaltene Snapshot wird *Commit* genannt. Git speichert für jeden Commit einen Pointer (dt.: Verweis) auf seine(n) Vorgänger, sodass stets die Änderungshistorie nachverfolgt werden kann (Chacon und Straub 2014, Kap.3). Aus diesen Referenzen kann eine Baumstruktur entstehen, da unterschiedliche Commits auf denselben Vorgänger zeigen können.

[9] engl.: **D**istributed **V**ersion **C**ontrol **S**ystem

2.5.3 Branch

Bei der Softwareentwicklung mit Git werden *Branches* verwendet. Branches sind Pointer, die auf einen bestimmten Commit zeigen. Wenn auf einem bestimmten Branch ein neuer Commit erfolgt, bewegt sich der Pointer des Branches automatisch auf den aktuellen Commit (Chacon und Straub 2014, Kap.3).

Die Voreinstellung von Git nennt den Hauptbranch eines Repositorys `master`. Um einen neuen Branch zu erstellen, wir der Befehl `git branch` verwendet. Zwischen verschiedenen Branches wird gewechselt, indem der Befehl `git checkout` ausgeführt wird. Wenn zwei verschiedene Branches auf unterschiedliche Commits verweisen, bedeutet das Wechseln zwischen ihnen, dass sich die Dateien im Working-Directory verändern (Chacon und Straub 2014, Kap.3).

Branches, die nicht auf der lokalen Maschine liegen, nennt man *Remote-Branches*. Wenn ein Projekt von einem solchen Remote-Repository (kurz: Remote, dt. etwa: entfernte Ablage) auf einem Server geklont wurde, speichert Git automatisch alle Referenzen, die in dem Remote liegen. Dies ermöglicht es, die Veränderungen auf Remote-Branches mit `git fetch` abzurufen. Standardmäßig benennt Git das Remote als `origin`. Lokale Branches können mit `git push` in das Remote kopiert werden, sodass auch andere mit Zugriffsrechten auf das Repository diese Branches sehen können (Chacon und Straub 2014, Kap.3).

Lokale Branches können Remote-Branches nachverfolgen (engl.: track). Bspw. trackt der lokale `master`-Branch i.d.R. den `origin/master`-Branch auf dem Server. Wenn auf einem lokalen Branch Veränderungen vorgenommen wurden, können diese mit dem Befehl `git push` auf den getrackten Remote-Branch geschoben werden. Mit `git pull` lassen sich die Veränderungen auf dem getrackten Remote-Branch direkt in den lokal aktuellen Branch (engl.: current branch) integrieren (Chacon und Straub 2014, Kap.3). Hierbei können Merge-Konflikte auftreten.

2.5.4 Merge und Merge-Konflikt

Sogenannte Feature-Branches ermöglichen das Entwickeln von Erweiterungen, von Lösungen für Programmfehler (engl.: Bugfixes) und von anderen Änderungen, ohne den Hauptbranch zu verändern. Wenn eine Veränderung, die in einem Branch fertiggestellt wurde, in einen anderen Branch – bspw. den Hauptbranch– integriert werden soll, wird ein *Merge* durchgeführt.

Um den Merge durchzuführen, verwendet Git drei Snapshots: den gemeinsamen Vorgänger (engl.: common ancestor) der verschiedenen Branches und die unterschiedlichen Commits, auf die die Pointer der Branches jeweils zeigen. Aus diesen Informationen wird ein Merge-Commit erstellt, der zwei Commits als Vorgänger hat, weil die zwei Branches zusammengeführt wurden (Chacon und Straub 2014, Kap.3).

Wenn auf beiden Branches dieselbe Datei an derselben Stelle verändert wurde, kommt es bei diesem Vorgang zu einem sogenannten *Merge-Konflikten*. Git unterbricht daraufhin den Merge-Prozess und der Merge-Konflikt muss händisch behoben werden. Wenn die Konflikte in den einzelnen Dateien beseitigt wurden, können diese mit `git add` gestaged werden, womit Git die Konflikte als behoben erkennt. Der Merge wird danach mit dem Befehl `git commit` abgeschlossen (Chacon und Straub 2014, Kap.3).

2.6 Releaseprozesse

Ein wichtiger Schritt in der Softwareentwicklung ist das Veröffentlichen der entwickelten Software. Die Version einer Software, die an Nutzende bzw. Kund:innen ausgeliefert wird,

nennt man ein *Release* (Sommerville 2018, Kap.25).

Ein Release enthält nicht nur den ausführbaren Quelltext, sondern kann bspw. auch Konfigurations-, Datendateien, ein Installationsprogramm und die Dokumentation der Software enthalten (Sommerville 2018, Kap.25).

Abseits von Programmen, die ausschließlich für den privaten Gebrauch geschrieben werden, ist ein *Releaseprozess* (bzw. Veröffentlichungsprozess) nötig, um eine Version einer Software für potenzielle Nutzer:innen verfügbar zu machen.

Die für die Veröffentlichung verwendeten Vorgehensweisen und Techniken können je nach Softwareentwicklungsprojekt stark variieren(Michlmayr u. a. 2007; Tsay u. a. 2011; Silva u. a. 2016). Sie sind davon abhängig, wohin die Software ausgeliefert wird, was dafür nötig ist, welche Artefakte in welcher Form ausgeliefert werden, wie oft ein Release entstehen soll etc.

In den folgenden Unterabschnitten soll das Feld der Releaseprozesse beschrieben werden, indem Beschreibungen der beiden Themengebiete Release-Management und Release-Engineering zitiert, drei Release-Management-Strategien wiedergegeben, in der OSS-Entwicklung von Michlmayr u. a. identifizierte Werkzeuge und Praktiken des Release-Managements dargestellt, Phasen eines Releasezyklus präsentiert und Nightly Builds als eine Variante von häufigen Releases vorgestellt werden.

2.6.1 Release-Management und Release-Engineering

Das *Release-Management* (RelMan, dt.: Veröffentlichungsverwaltung) bzw. das *Release-Engineering* (RelEng, dt.: Veröffentlichungstechnik) befassen sich mit den Aktivitäten, die zwischen der Softwareentwicklung und der tatsächlichen Nutzung eines Softwareprodukts ausgeführt werden (Adams u. a. 2013).

Laut Sommerville gehören zum RelMan die folgenden Tätigkeiten: die Entscheidung über den Zeitpunkt der Releases, die Dokumentation eines jeden Release und die Vorbereitung aller Informationen für die Verteilung (Sommerville 2018, Kap.25).

Allerdings wird der Rolle eines Release-Managers von Michlmayr u. a. die Verantwortlichkeit für die Erstellung und Veröffentlichung eines qualitativ hochwertigen Release zugeordnet (Michlmayr u. a. 2007), womit in den Bereich des RelMans mehr Aufgaben fielen, als die von Sommerville beschriebenen.

Laut Michlmayr u. a. muss ein Release-Manager mit einer großen Gruppe von an der Entwicklung beteiligten Personen interagieren, technische Sachverhalte verstehen und die Abläufe planen und koordinieren können (Michlmayr u. a. 2007). Je nach Größe des Projekts könne die Rolle sehr unterschiedliche Ausprägungen haben. Bspw. würden sich die Aufgaben eines Release-Managers in kleinen Projekten meist auf die Vorbereitung eines Release in verschiedenen, verteilbaren Formaten, die Erstellung von Releasenotes und die tatsächliche Bereitstellung der Software beschränken. Wohingegen dieser Rolle in größeren Projekten v.a. koordinatorische Aufgaben zugeteilt werden würden, wie die Sicherstellung, dass verschiedene Teile der Software zum gleichen Zeitpunkt fertig seien und dass alle Entwickelnden auf das gemeinsame Ziel ausgerichtet sind, ein stabiles Release zu erstellen (Michlmayr u. a. 2007).

Andere Autoren bezeichnen dieselbe Rolle mit dem Titel Release-Engineer (Tsay u. a. 2011; Spinellis 2018; Rahman 2015). Je nach Projekt wird die Rolle von einer Person oder einem Team ausgefüllt (Tsay u. a. 2011).

Von Dyck u. a. wurde folgende Definition für RelEng als eine Unterdisziplin der Softwaretechnik vorgeschlagen:

> Release engineering is a software engineering discipline concerned with the development, implementation, and improvement of processes to deploy high-quality software reliably and predictably. (Dyck u. a. 2015)

Zu Deutsch etwa: Release-Engineering ist eine Disziplin der Softwaretechnik, die sich mit der Entwicklung, Implementierung und Verbesserung von Prozessen zur zuverlässigen und vorhersagbaren Bereitstellung hochwertiger Software befasst.

Die Verbesserung der Prozesse beinhalte auch, dass der Informationsfluss zwischen den beteiligten Personen verbessert werden würde (Dyck u. a. 2015).

Adams u. a. sagen, dass der Fokus des RelEng darauf liege, eine Pipeline aufzubauen, welche Quellcode in ein integriertes, kompiliertes, verpacktes (engl.: packaged), getestetes und signiertes Produkt umwandelt, das für die Veröffentlichung bereit ist. Der Input dieser Pipeline sei dabei der von Entwickelnden geschriebene Quellcode zur Erstellung eines Produkts oder zur Modifizierung eines bestehenden Produkts (Adams u. a. 2015).

In traditionellen Softwareentwicklungsmethodiken wie dem Wasserfall- und dem Spiralmodell wird RelEng normalerweise als Teil der Phase *Betrieb und Wartung* angesehen (Tsay u. a. 2011).

2.6.2 Release-Management-Strategien

Poo-Caamaño identifiziert drei unterschiedliche Release-Management-Strategien:

- Funktionsgesteuerte Entwicklung (engl.: feature driven development)
- Zeitbasierte Entwicklung (engl.: time based development)
- Qualitätsbasierte Entwicklung (engl.: quality based development)

Je nach Strategie basiert die Entscheidung zur Veröffentlichung eines neuen Release demnach entweder darauf, dass die Entwicklung bestimmter Funktionen abgeschlossen wurde, dass ein bestimmtes Datum als Releasezeitpunkt festgelegt wurde oder dass die implementierten Neuerungen ein für die Veröffentlichung benötigtes Minimum an Qualität haben (Poo-Caamaño 2016).

Es wird gesagt, dass maximal zwei dieser Strategien gleichzeitig verfolgt werden können und deswegen beim RelMan priorisiert werden muss, auf welche der Fokus gelegt wird (Poo-Caamaño 2016). Bspw. müsse innerhalb eines Projekts der Featureumfang reduziert werden, um ein Release von vorgegebener Qualität zu einem festgelegten Datum zu veröffentlichen (Poo-Caamaño 2016).

2.6.3 Werkzeuge und Praktiken

Michlmayr u. a. identifizieren Werkzeuge und Praktiken, die innerhalb des RelMans in FOSS[10]-Projekten eingesetzt werden.

Neben dem Nutzen von Systemen zur Versionsverwaltung und Fehlerverfolgung (engl.: bug tracking) (Michlmayr u. a. 2007), werden die folgenden Release-Management-Praktiken aufgeführt:

- Freezing: Die Entwicklung von Neuerungen würde eingestellt, um den Fokus auf die Entfernung von Fehlerhaftem und auf die Veröffentlichung des Release zu legen (Michlmayr u. a. 2007).

[10] **F**ree and **O**pen **S**ource **S**oftware

- Scheduling: Bei Projekten mit zeitbasierten Release-Strategien sei der Zeitplan eine unerlässliche Komponente (Michlmayr u. a. 2007).
- Establishing milestones: Diese seien oft lose definiert und es gibt in FOSS-Projekten keine Garantie auf Erfüllung der Meilensteine, da die Entwickelnden freiwillig arbeiten (Michlmayr u. a. 2007).
- Setting deadlines: Viele Projekte setzten Fristen, allerdings seien sie nicht immer effektiv aufgrund der Freiwilligkeit von FOSS-Entwickelnden (Michlmayr u. a. 2007).
- Building on different architectures: Es sei hilfreich, Software als Teil der Teststrategie auf unterschiedlichen Hardwaresystemen zu bauen (Michlmayr u. a. 2007).
- User testing: Die wichtigsten Erkenntnisse würden durch das Feedback von Nutzenden der Software gewonnen (Michlmayr u. a. 2007).
- Following a release checklist: Um sicherzustellen, dass alle für ein Release benötigten Tätigkeiten aufgeführt werden, nutzten viele Projekte eine Checkliste (Michlmayr u. a. 2007).
- Holding a post-release review: Auch wenn wenige Projekte einen solchen formellen Rückblick nach einem Release durchführten, würden auf informellen Wegen Diskussionen zum Release und aufgetretenen Problemen bei der Veröffentlichung stattfinden (Michlmayr u. a. 2007).

Silva u. a. beschreiben drei Strategien, die von Entwickelnden in OSS-Projekten verfolgt wurden, um ihre Software-Release-Praktiken agiler zu gestalten. Diese seien die Anwendung einer zeitbasierten Release-Strategie, das Nutzen von automatisierten Testwerkzeugen für die Umsetzung eines effektiven Qualitätsmanagements und der Einsatz eines Continuous-Delivery-Prozesses (s. 2.7.2 Continuous Delivery/Deployment) (Silva u. a. 2016).

2.6.4 Phasen eines Releasezyklus

Rahman teilt den Releasezyklus, also die Zeit, die zwischen den Veröffentlichungen von zwei aufeinanderfolgenden Releases liegt, in die Phasen *Entwicklung* und *Stabilisierung* ein. Während der Entwicklungsphase würden die Entwickelnden an neuen oder an bestehenden Features arbeiten (Rahman 2015). Ein Release-Engineer trüge in der Stabilisierungsphase die Verantwortung dafür, die Features zu sammeln und zu stabilisieren, bevor sie veröffentlicht werden (Rahman 2015).

Die von Michlmayr u. a. beschriebene Praktik des Freezings ist in der Stabilisierungsphase anzusiedeln.

2.6.5 Nightly Builds

Ein *Nightly Build* wird traditionell nachts, wenn der Quellcode i.d.R. nicht bearbeitet wird, durchgeführt, um eine Software in der Entwicklungsphase regelmäßig automatisiert zu bauen und ggf. zu testen (Shake Technologies, Inc. 2023). Die Software wird mit allen am Tag zuvor eingecheckten Neuerungen in einer geeigneten Build-Umgebung gebaut. Die Erstellung von Nightly Builds dauert u.U. länger als das Durchlaufen einer Continuous-Integration-Pipeline (s. 2.7.1 Continuous Integration), wenn sie umfangreiche Tests beinhaltet.

Nightly Builds können entweder nur für die interne Entwicklung genutzt werden oder zusätzlich auch veröffentlicht werden. Sie können dem Entwicklungsteam dazu dienen, zum einen in der Entwicklung entstandene Fehler früher zu finden und zum anderen sicherzustellen, dass die zum Bauen und Testen genutzte Infrastruktur noch funktioniert (Witko

2019). Desweiteren können Nightly Builds als solche veröffentlicht werden, um Nutzenden die Möglichkeit zu geben, zwar potenziell instabile und weniger getestete, aber dafür neuere Versionen einer Software auszuprobieren[11].

2.7 CI/CD

Wenn die Entwicklung von Software auf verschiedene Teams bzw. Personen aufgeteilt wird, kann es schnell zu Konflikten zwischen den jeweils neu erstellten Quelltextsegmenten kommen. Brun u. a. haben herausgefunden, dass etwa bei einem Fünftel aller Merges in Open-Source-Projekten Konflikte auftreten (Brun u. a. 2011). Sei es, weil an derselben Stelle Zeilen von mehreren Akteuren unterschiedlich verändert wurden oder weil ein neues Segment eine ältere Implementation eines Teils der Software benötigt, der indes refaktorisiert wurde und nun auf eine andere Art und Weise funktioniert. Solche Probleme sind wahrscheinlicher, wenn es sich um ein großes Softwareprojekt handelt, an dem viele Menschen arbeiten (Menezes u. a. 2020).

Diese Konflikte werden Integrationsprobleme (engl.: integration problems) bzw. Merge-Konflikte (engl.: merge conflicts) genannt und sie treten i.d.R. am Ende der Entwicklungsphase auf, wenn alle neu entwickelten Funktionen zusammengeführt werden sollen. Denn je länger die Entwicklungen auf einem Branch nicht in den Hauptbranch zurückgeführt wird, desto höher ist die Wahrscheinlichkeit eines Konflikts (Menezes u. a. 2020). Und je länger diese Entwicklungsphase ist bzw. je umfangreicher die neuen Funktionen sind, desto unberechenbarer werden Merge-Konflikte (Menezes u. a. 2020) – sowohl ihre Komplexität betreffend, als auch zeitlich (Nelson u. a. 2019).

Aus diesem Grund ist es offensichtlich ratsam einen strategischen Umgang zu finden, der möglichst zuverlässig verhindert, dass der Mergeprozess unvorhersehbar kompliziert und zeitaufwändig wird. *Continuous Integration* (CI, dt.: kontinuierliche Integration) ist eine solche Strategie, bei der Änderungen mindestens täglich in die Codebasis integriert werden und jeder Merge durch Tests und automatisierte Builds überprüft wird (Fowler 2024). Die Überzeugung dabei ist, dass der Aufwand und die Kosten von Mergeprozessen insgesamt dadurch geringer werden, dass sie ständig durchgeführt werden. Die ständige Durchführung bedingt, dass kleinere Änderungen integriert werden, was wiederum die Komplexität der Merges reduziert (Thönes und Humble 2015; Fowler 2011).

Auch das Ausliefern (engl.: deployment) von Software kann ein langwieriger Prozess sein (Humble 2017a). *Continuous Delivery* bzw. *Continuous Deployment* (CD, dt.: kontinuierliches Abliefern bzw. Bereitstellen) ist ein Konzept, welches als eine Art Erweiterung von CI auch diesen Vorgang zuverlässiger gestalten soll, indem er ständig ausgeführt wird bzw. ausführbar ist (Humble 2017a). Chen bspw. berichtet von einer schnelleren Auslieferung seit der Einführung von CD (Chen 2015).

In den folgenden Abschnitten werden die beiden DevOps[12]-Praktiken CI und CD noch etwas genauer beschrieben.

2.7.1 Continuous Integration

CI ist eine DevOps-Praktik, die in Softwareentwicklungsprojekten genutzt werden kann, um das Integrieren von Software stabiler zu machen und zu beschleunigen (Fowler 2024). Mit

[11] Bsp.e für Veröffentlichungen von Nightly Builds von Android Studio `https://developer.android.com/studio/nightly` und Brave `https://brave.com/download-nightly/`.

[12] Software **Dev**elopment und IT **Op**eration**s**

Integrieren ist hierbei das Mergen von Feature- oder Bugfix-Branches auf den Hauptbranch gemeint.

Es wird kontinuierlich auf den Hauptbranch des gemeinsamen Remote-Repositorys gepusht. Dies sollte mindestens täglich passieren (Humble 2017b; Fowler 2024).

Im Remote wird der Code sofort automatisiert gebaut und getestet. Die Tests sollten umfangreich genug sein, dass darauf vertraut werden kann, dass weiterhin alles funktioniert. Gleichzeitig sollten sie schnell genug laufen, dass ein zügiges Feedback möglich ist (Humble 2017b; Fowler 2024).

Wenn das Remote-Repository so aufgesetzt wird, sieht das Entwicklerteam sofort, falls etwas nicht korrekt gebaut wird oder Tests nicht erfolgreich bestanden werden. In diesem Fall sollten alle anderen Entwicklungen unterbrochen werden, um den Fehler zu beheben (Humble 2017b; Fowler 2024).

In Abbildung 2.2 ist zu sehen, wie der Push auf das Remote auslöst, dass die Software automatisiert gebaut und dann automatisch getestet wird. Sollte entweder beim Bauen oder beim Testen etwas fehlschlagen, muss der Code überarbeitet und erneut gepusht werden.

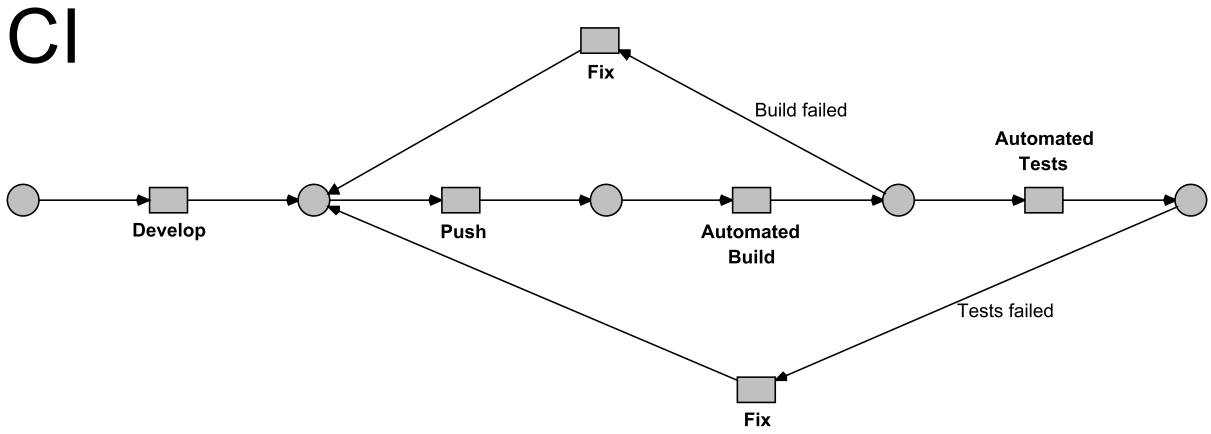


Abbildung 2.2: Beispiel für ein Schema einer einfachen CI-Pipeline

CI soll dazu führen, dass weniger komplizierte Integrationskonflikte entstehen (Fowler 2024). Das wird dadurch sichergestellt, dass der Code ständig um kleine Schritte erweitert wird, die auch mit dem Rest der Codebasis funktionieren. Im Gegensatz dazu steht der alte Ansatz, bei dem u.U. erst kurz vor einem Sprintende oder Release lauter Fehler bei einem großen Merge auffallen.

Insgesamt soll CI zu höherer Softwarequalität führen, da die Personen im Entwicklungsteam dazu animiert werden, sorgfältiger und an kleinen Inkrementen zu arbeiten (Fowler 2024).

Zusammenfassend gesagt, werden durch die Anwendung von CI Fehler innerhalb einer Änderung schneller entdeckt und sie werden sofort adressiert. Außerdem wird zügiger bemerkt, wenn neue Bestandteile nicht zusammenpassen. Gleichzeitig ist immer ein aktueller, lauffähiger Stand der Software verfügbar.

2.7.2 Continuous Delivery/Deployment

Continuous Delivery bzw. Continuous Deployment (CD) sind wie CI auch DevOps-Praktiken. Je nach Kontext steht das „D“ in CD für Delivery (dt.: Lieferung) oder Deployment (dt.: Bereitstellung), wobei Continuous Deployment als eine Erweiterung von Continuous Delivery angesehen werden kann, die diese mit einschließt.

Ein Prinzip von CD ist, dass die ständige Auslieferbarkeit (engl.: deployability) der Software wichtiger ist als neue Funktionen (Fowler 2013). Der Fokus des Entwicklungsteams wird also u.U. auf diese Auslieferbarkeit verschoben, wenn er vor der Implementierung von CD noch nicht darauf lag.

Die Idee, dass die Software zu jedem Zeitpunkt auslieferbar ist, bedingt, dass eine sogenannte Push-Button-Bereitstellung (push button, zu dt. etwa: Knopfdruck) jeder Version der Software möglich ist (Fowler 2013). Es sollte automatisierte Rückmeldungen dazu geben, ob die Systeme production-ready (dt.: ausbringungsbereit) sind (Fowler 2013), indem die Software z.B. automatisiert in einem Staging-Environment (dt.: Staging-Umgebung) getestet wird.

Wenn von Deployment gesprochen wird, erweitert CD sich darum, dass das Produkt auch automatisiert ausgeliefert und in die Produktionsumgebung übergeben wird (Fowler 2013).

In Abbildung 2.3 ist ein Beispiel für eine einfache CD-Pipeline zu sehen, in der veränderter Code, der die CI erfolgreich abgeschlossen hat, weitere umfassendere Acceptance Tests durchläuft. Wenn der neue Code alle automatisierten Testphasen bestanden haben, ist sie bereit für die automatisierte Auslieferung in eine Testumgebung (engl.: staging environment). Sollte das Ausliefern in die Staging-Umgebung funktioniert haben, kann die Auslieferung der Software inkl. der Neuerungen in die Produktionsumgebung vorgenommen werden. Bei Continuous Delivery passiert dieser Schritt händisch; bei Continuous Delivery ist auch dieser letzte Schritt automatisiert.

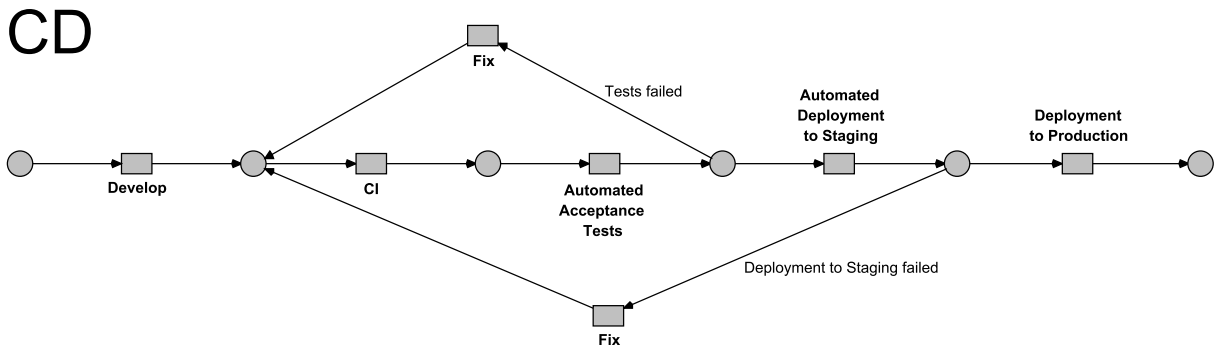


Abbildung 2.3: Beispiel für ein Schema einer einfachen CD-Pipeline

Zusammenfassend lässt sich sagen, dass durch CD Folgendes verbessert wird (Fowler 2013):

- Insgesamt wird das Risiko von Fehlern und Bugs beim Ausliefern in die Produktion kleiner.
- Schnellerer und glaubwürdiger Entwicklungsfortschritt wird sichtbar gemacht. Das bedeutet auch, dass neue Funktionen nicht nur prototypisch fertig sind, sondern tatsächlich production-ready vorliegen.
- Wenn zusätzlich automatisch deployed wird, gibt es außerdem häufigere Rückmeldungen von denen, die die Software nutzen. Somit kann schneller ermittelt werden, ob Änderungen so angenommen werden, wie sie gedacht waren, oder etwa gar nicht gewünscht sind.

2.8 GitLab

GitLab ist eine Webanwendung, welche für die Versionsverwaltung mit Git verwendet werden kann. Diese kann über die grafische Benutzeroberfläche (GUI[13]) im Browser genutzt werden, es gibt aber auch eine Programmierschnittstelle (API[14]).

Bei der Entwicklung von RENEW wird GitLab genutzt; das entsprechende GitLab-Projekt wird auf Servern des Rechenzentrums der Universität Hamburg gehostet.

Neben der Möglichkeit, Git-Repositories zu hosten und zu verwalten, bietet GitLab noch weitere Funktionalitäten, u.a. auch die Einbindung von CI/CD-Praktiken.

2.8.1 GitLab Merge-Requests

Ein *Merge-Request* ist ein Vorschlag, Änderungen aus einem Branch in einen anderen sogenannten Zielbranch zu integrieren (GitLab Inc. 2024n).

Wird ein Merge-Request in einem GitLab-Projekt geöffnet, beinhaltet dieser u.a. eine Beschreibung des Requests, die Codeänderungen, die in den Zielbranch integriert werden sollen, eine Auflistung der Commits, in denen die Änderungen vorgenommen wurden, Informationen über die CI/CD-Pipelines und eine Kommentarspalte, in der am Projekt beteiligte Personen über die Änderungen diskutieren können (GitLab Inc. 2024n).

In GitLab-Projekten können bestimmte Branches als `protected` (dt.: geschützt) markiert werden. Auf diesen Branches können nur Personen mit bestimmten Rechten Änderungen vornehmen. Das bedeutet auch, dass nur diese Personen die Änderungen eines Merge-Requests in einen geschützten Zielbranch mergen können (GitLab Inc. 2024m).

Für Merge-Requests kann festgelegt werden, dass sie eine Freigabe (engl.: approval) benötigen, bevor sie integriert werden können. Auf diese Art kann die Durchführung von Codereviews verpflichtend festgesetzt werden (GitLab Inc. 2024l).

Falls innerhalb eines Merge-Requests Merge-Konflikte existieren, wird er automatisch von GitLab blockiert, bis die Konflikte aufgelöst wurden (GitLab Inc. 2024k).

2.8.2 GitLab CI/CD

GitLab CI/CD ist ein von GitLab bereitgestelltes Werkzeug, welches genutzt werden kann, um CI/CD-Praktiken in einem auf einem GitLab-Server gehosteten Projekt zu realisieren.

Um GitLab CI/CD verwenden zu können, sind mindestens ein Runner und eine YAML[15]-Datei namens `.gitlab-ci.yml` nötig.

GitLab Runner

Die Anwendung *GitLab Runner* muss auf einem Server installiert werden und hier können dann einzelne Runner (dt.: Läufer) registriert werden. Das passiert, um die Kommunikation zwischen der eigenen GitLab-Instanz und der Maschine, auf der GitLab Runner installiert ist, herzustellen (GitLab Inc. 2024h).

Runner implementieren Executor (dt.: Ausführer), die während der Registrierung gewählt werden. Diese können z.B. Shell Executor, Virtual Machine Executor oder Docker Executor sein. Das universitäre Softwareprojekt RENEW verwendet Docker Executor (Feldmann

[13] engl.: **G**raphical **U**ser **I**nterface

[14] engl.: **A**pplication **P**rogramming **I**nterface

[15] YAML steht für „**Y**AML **A**in't **M**arkup **L**anguage“ (The YAML Project 2021) und ist eine Auszeichnungssprache, die häufig für Konfigurationsdateien genutzt wird (Feldmann 2019, S.80).

2019, S.15). Die Runner führen später die einzelnen Jobs der CI-Pipeline aus bzw. die Skripte, die in den Jobs beschrieben werden (GitLab Inc. 2024h).

In der YAML-Datei kann für jeden einzelnen Job ein spezifischer Runner ausgewählt werden, indem die Tags angegeben werden, die auch dem jeweiligen Runner gegeben wurden (GitLab Inc. 2024h).

Wenn ein Docker Executor verwendet wird, wird das entsprechende Docker Image für den Docker Container benötigt. Das gewünschte Image kann global für alle Jobs in der YAML-Datei angegeben werden oder bei Bedarf für jeden Job einzeln definiert werden (GitLab Inc. 2024f).

`.gitlab-ci.yml`

In GitLab CI/CD heißt die YAML-Datei, mit der die CI/CD-Prozesse konfiguriert werden, standardmäßig `.gitlab-ci.yml` und liegt, sofern es nicht anders spezifiziert wurde, im Stammverzeichnis des jeweiligen GitLab-Projekts (GitLab Inc. 2024e).
In dieser Datei wird unter anderem Folgendes definiert (GitLab Inc. 2024c):

- welche Pipelines, Stages, Jobs, Skripte, Caches, Artefakte und Variablen es gibt, jeweils mit ihren entsprechenden Konfigurationen,
- welche Abhängigkeiten zwischen den einzelnen Jobs bestehen,
- welche Sequenzen und Nebenläufigkeiten innerhalb der CI/CD-Vorgänge auftreten sollen bzw. dürfen,
- ob andere Konfigurationsdateien und -templates genutzt werden, was mit dem Schlüsselwort `include` gekennzeichnet wird, und
- in welche Umgebung die Software ausgeliefert wird.

Pipelines (GitLab Inc. 2024b)

Die *Pipelines* sind die übergeordnete Struktur für CI/CD-Prozesse und beinhalten *Jobs* (dt.: Aufgaben) und *Stages* (dt.: Abschnitte). Sie werden i.d.R. nach jedem push zum Remote--Repository automatisch ausgeführt, können aber auch zu bestimmten Zeiten geplant oder getriggert werden. Außerdem können Jobs in einer Pipeline mit Hilfe von Regeln aus- oder eingeschlossen werden, sodass z.B. ein Job nur ausgeführt wird, wenn die Pipeline durch einen Merge Request ausgelöst wurde.

Eine typische Pipeline könnte aus den folgenden vier Stages bestehen: *build*, *test*, *stage* und *production*. In der build-Stage könnte ein Job sein, der den Code kompiliert. In der test-Stage könnten Jobs sein, die den kompilierten Code testen. In der staging-Stage könnte ein Job sein, der den Code in die Staging-Umgebung ausliefert. Und in der production-Stage könnte ein Job sein, der den Code, der alle vorherigen Stages erfolgreich durchlaufen hat, in die Produktionsumgebung ausliefert.

Insgesamt können Pipelines sehr individuell konfiguriert werden. Eine Pipeline kann bspw. auch lediglich ein Latexdokument zu einer PDF[16]-Datei kompilieren und als Artefakt speichern, sodass sie zu einem späteren Zeitpunkt heruntergeladen werden kann.

Stages (GitLab Inc. 2024c)

Stages definieren, wann Jobs ausgeführt werden. Z.B. kann eingestellt werden, dass Stages, in denen test-Jobs ausgeführt werden, erst vollzogen werden, nachdem Stages, in denen der Code kompiliert wurde, bereits durchlaufen wurden.

[16] **P**ortable **D**ocument **F**ormat

Sie werden nacheinander ausgeführt und die Reihenfolge der Ausführung wird durch ihre Reihenfolge in der YAML-Datei festgelegt. In der Regel wird eine Stage nur ausgeführt, wenn alle vorhergegangenen Stages erfolgreich abgeschlossen wurden.

Jobs (GitLab Inc. 2024c; GitLab Inc. 2024j)

In den Jobs wird definiert, was gemacht werden soll, indem hier Skripte angegeben werden. Die auszuführenden Skripte legen fest, wie der Code z.B. gebaut oder getestet werden soll.

Jobs werden von einem Runner ausgeführt. Sie haben immer eine Stage und Jobs, die zu der gleichen Stage gehören, werden parallel ausgeführt, sofern dafür genügend nebenläufige Runner existieren. Sobald alle Jobs einer Stage erfolgreich abgeschlossen wurden, fährt die Pipeline mit der nächsten Stage fort. Falls irgendein Job fehlschlägt, dann wird i.d.R. die nächste Stage nicht ausgeführt und die gesamte Pipeline wird abgebrochen.

In welcher Pipeline welche Jobs ausgeführt werden, kann in der YAML-Datei mit den Schlüsselwörtern `rules`, `only` und `except` konfiguriert werden. Außerdem kann unter dem `default`-Schlüsselwort festgelegt werden, ob bestimmte Dinge für jeden Job vor bzw. nach dem Skript ausgeführt werden sollen. Das geht auch für jeden einzelnen Job mit den Schlüsselwörtern `before_script` und `after_script`.

Cache (GitLab Inc. 2024d)

Die Verwendung des Caches (dt.: Zwischenspeicher) erlaubt eine schnellere Ausführung von Jobs, indem Abhängigkeiten wie etwa aus dem Internet heruntergeladene Packages (dt.: Pakete) gespeichert werden. Diese Cache-Dateien werden im Runner abgelegt.

Die Defaulteinstellung hierbei ist, dass Jobs vor ihrer Ausführung aus dem Cache pullen und danach Veränderungen in den Cache pushen. Später ausgeführte Pipelines und Jobs können auf den Cache zugreifen, indem sie denselben mit einem Tag versehenen Runner nutzen oder denselben Cache-Schlüssel – z.B. eine CI/CD-Variable, die den Branch referenziert – verwenden.

Artefakte (GitLab Inc. 2024a; GitLab Inc. 2024i)

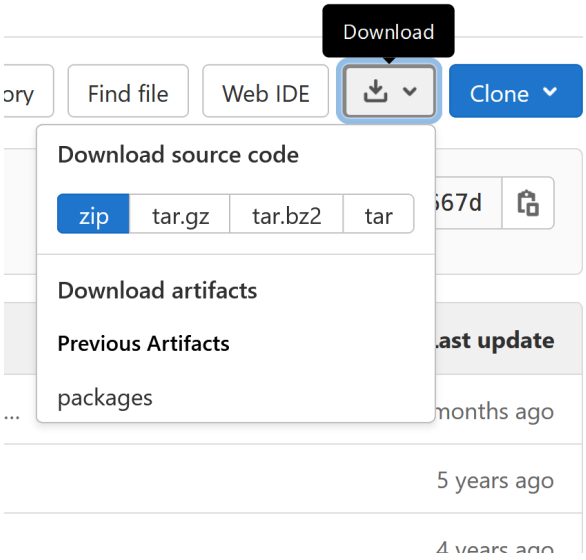


Abbildung 2.4: Download über die GUI

Artefakte (engl.: artifacts) können von einem Job generiert werden. Es ist möglich, diese daraufhin in GitLab zu speichern und auch über die GUI von GitLab (s. Abbildung 2.4) oder über die API herunterzuladen. Job-Artefakte können außerdem genutzt werden, um z.B. Ergebnisse, die in der build-Stage entstanden sind, oder andere generierte Dateien für spätere Stages derselben Pipeline verfügbar zu machen.

Standardmäßig verfallen Artefakte nach 30 Tagen, aber *latest artifacts*, also die zuletzt von einer Pipeline generierten Artefakte, verfallen nicht, sondern bleiben erhalten, bis die Pipeline erneut angestoßen wird.

Die Artefakt-Datei eines Jobs, welche eine ZIP-Datei ist, kann dynamisch benannt werden – dies könnte z.B. der branch-Name aus der CI/CD-Variable sein, allerdings sollte hierbei CI_COMMIT_REF_SLUG statt CI_COMMIT_REF_NAME verwendet werden, falls Branchnamen mit Schrägstrichen im Projekt existieren.

Variablen (GitLab Inc. 2024g)

Es existieren vordefinierte CI/CD-Variablen, die genutzt werden können, und es ist auch möglich individuelle Variablen in der YAML-Datei anzulegen. Bspw. gibt es viele, die über den Commit Auskunft geben, der die Pipeline gestartet hat – wie z.B. die vordefinierte Variable CI_COMMIT_REF_NAME –, über den Job und die Pipeline selbst, über das Projekt, zu dem die Pipeline gehört, über den Runner, den Server usw.

Diese Variablen können genutzt werden, um das Verhalten von Jobs und Pipelines zu kontrollieren. Außerdem können Werte mit ihnen als Schlüssel gespeichert werden, die später weitergenutzt werden sollen. Des Weiteren lässt sich durch ihre Verwendung verhindern, bestimmte Werte in der YAML-Datei fest einprogrammieren (engl.: hardcode) zu müssen.

Kapitel 3

Anforderungen

Dieses Kapitel beschreibt die Anforderungen an diese Arbeit. Zum einen wird hierfür in den Abschnitten „Durchführung von Releases“, „Wissensbündelung“, „Dokumentation von Herausforderungen“ und „Betrachtung möglicher Verbesserungen“ festgehalten, was die Ziele dieser Arbeit sind; zum anderen wird im letzten Abschnitt zusätzlich gesagt, wo die Grenzen dieser Arbeit liegen.

3.1 Durchführung von Releases

Begleitet durch diese Arbeit sollen neue Versionen von RENEW veröffentlicht werden. Hierfür ist es nötig, alle Schritte des Releaseprozesses durchzuführen.

Insbesondere die Veröffentlichung einer modularen Variante von RENEW wird im Rahmen dieser Arbeit zum ersten Mal realisiert, was bedeutet, dass der bisherige Veröffentlichungsprozess für klassische Versionen adaptiert werden muss. Die jeweils aktuelle Version der modularen und klassischen Varianten sollen nebeneinander auf RENEWs Webseite angeboten werden.

Die Durchführung von Releases bildet die Grundlage für die darauffolgende Wissensbündelung und die Dokumentation der Herausforderungen.

3.2 Wissensbündelung

Der Releaseprozess von RENEW ist bisher intransparent. Das Wissen über die Arbeitsschritte, die für die Veröffentlichung einer neuen Version der Software nötig sind, ist nicht bzw. nicht gebündelt schriftlich festgehalten.

Dieser Umstand ist zum Teil ursächlich dafür, dass Veröffentlichungen selten durchgeführt werden. Wenn eine Durchführung bevorsteht, bestehen große Unsicherheiten über den Ablauf. Diese verzögern den Vorgang zusätzlich.

Aus diesem Grund ist es vonnöten, den Prozess zu dokumentieren und für nachfolgende Iterationen der Veröffentlichung bereitzustellen. Es sollen Konventionen und Vorgehensweisen festgehalten werden, welche innerhalb einiger Projektdurchläufe erarbeitet wurden. Durch das Explizieren und Strukturieren des Vorgehens soll eine Nachhaltigkeit der Abläufe geschaffen werden.

Die Dokumentation soll außerdem beinhalten, welche Akteure und Techniken am Veröffentlichungsprozess von RENEW beteiligt sind und welche Artefakte zu einer Veröffentlichung gehören.

3.3 Dokumentation von Herausforderungen

Die Probleme bei der Veröffentlichung von RENEW sollen dokumentiert werden. Das heißt, dass der Veröffentlichungsprozess kritisch betrachtet werden soll, um daraufhin Vorschläge für mögliche Anpassungen und Automatisierungen für diesen Prozess machen zu können.

Die Betrachtung erfolgt auf Basis von Erfahrungen, die während der Durchführung der Releases gesammelt wurden, und von durch Literatur dokumentierte Herausforderungen, die innerhalb von Releaseprozessen auftreten.

3.4 Betrachtung möglicher Verbesserungen

Die Veröffentlichung von Software lässt sich durch die technisch gestützte Automatisierung unterschiedlicher Arbeitsschritte für die entwickelnden Personen einfacher, zuverlässiger und schneller gestalten. Darüber hinaus können Konventionen und Leitfäden für Abläufe festgehalten werden, die durch ihre Wiederholbarkeit und die durch sie erzielte Vereinheitlichung der Prozesse eine gewisse Sicherheit bei der Durchführung der einzelnen Tätigkeiten erzielen.

Diese Arbeit untersucht Möglichkeiten zur Verbesserung des Veröffentlichungsprozesses von RENEW. Insbesondere werden neue Konventionen innerhalb des Releaseprozesses vorgeschlagen, Möglichkeiten für die Beseitigung organisatorischer und struktureller Hürden aufgezeigt und potentielle Automatisierungen bestimmt.

3.5 Grenzen dieser Arbeit

Diese Arbeit wird, basierend auf der Dokumentation des Veröffentlichungsprozesses und der Analyse von diesem, Vorschläge für mögliche Verbesserungen machen.

Die Umsetzung eines perfekten, vollautomatisierten Veröffentlichungsprozesses wird nicht angestrebt. Das bedeutet auch, dass diese Arbeit bspw. nicht die Implementation einer vollständig automatisierten Deployment-Pipeline oder die Veröffentlichung von Nightly Builds umsetzen wird.

Das Einführen von Metriken für die Quantifizierung von Releaseprozessen wäre nötig, um die Effizienz und Effektivität der Abläufe zu überprüfen.

Das Messen von Zuverlässigkeit, Schnelligkeit und Qualität ist nicht Fokus dieser Arbeit; vielmehr geht es darum, den Veröffentlichungsprozess zu explizieren und zu strukturieren, um eine Nachhaltigkeit der Abläufe sicherzustellen und daraufhin Möglichkeiten der Unterstützung durch Software, Konventionen etc. aufzuzeigen.

Innerhalb dieser Arbeit werden keine automatisierten Tests geschrieben oder Vorschläge für das Schreiben dieser erarbeitet.

Dies ist ein eigenes Thema, welches bereits in einer anderen Arbeit bearbeitet wurde (s. Al Shriteh 2022).

Kapitel 4

Organisation und Struktur der Veröffentlichungspraxis

Als Teil dieser Arbeit wurde der Veröffentlichungsprozess von den Versionen 2.6, 4.0 und 4.1 von RENEW innerhalb universitärer Projektarbeit begleitet. Die Versionen 2.6 und 4.0 wurden zusammen veröffentlicht, Version 4.1 wurde als Aktualisierung von 4.0 veröffentlicht.

Zur Einordnung, wie ein solcher Veröffentlichungsprozess für ein universitäres OSS-Projekt, welches umfangreich ist und bereits seit Jahrzehnten besteht, gestaltet werden kann, werden die gesammelten Erfahrungen in diesem und dem nachfolgenden Kapitel 5 wiedergegeben. Hierdurch soll gleichzeitig eine ausführliche Dokumentation des Releaseprozesses innerhalb der RENEW-Entwicklung erreicht werden. Während in Kapitel 5 alle Tätigkeiten beschrieben werden, die für die Durchführung eines Release nötig sind, legt dieses Kapitel den Fokus auf die folgenden Aspekte:

In „Voraussetzungen und Organisation eines Release“ wird beschrieben, wann ein Release entsteht, an welcher Stelle in den Phasen der Softwareentwicklung der RENEW-Releaseprozess einzuordnen ist und wie das durchführende Releaseteam aufgebaut ist.

Es folgt der Abschnitt „Versionierung und Aufbau von RENEWs Releases“, in welchem die Benennung der RENEW-Versionen, der Unterschied zwischen RENEWs zwei Produktlinien und die Struktur der Releases bzgl. des Umfangs, der zu veröffentlichenden Artefakte und der Lizenzbedingungen erklärt werden.

Danach wird in Abschnitt „Branching und Qualitätssicherung bei Änderungen auf dem Hauptbranch“ dargelegt, wie Änderungen auf den Hauptbranch durch festgelegte Branching-Konventionen und Qualitätssicherungsmaßnahmen abgesichert werden und deswegen als potenziell bereit für eine Veröffentlichung angesehen werden können.

Im Abschnitt „Buildmanagement und CI/CD von RENEW“ werden verschiedene Werkzeuge bzw. ihre Implementation innerhalb der Entwicklung von RENEW beschrieben, die gewisse Schritte im Releaseprozess automatisieren: die Buildtools Gradle und Ant, und GitLab CI/CD für die Bereitstellung der zu veröffentlichenden Artefakte.

4.1 Voraussetzungen und Organisation eines Release

Ein neues Release von RENEW wird angestrebt, wenn von Entwickelnden für die Veröffentlichung relevante Neuerungen erarbeitet wurden und der CCPO in Austausch mit den anderen CPOs und SoSMs (s. 2.4.1 Rollen) beschließt ein Release anzufordern.

Relevante Neuerungen können sowohl neue Plugins, neue Features innerhalb bestehender Plugins, Bugfixes oder Codewartung sein, als auch ausführlichere Dokumentation oder

umfangreichere Tests. Letztere sind für die Veröffentlichung neuer Versionen relevant, da RENEW eine OSS ist, deren Quelltext also auch veröffentlicht wird.

Ein Hauptentwickler hat vorgeschlagen, die folgenden Fragen vor dem Beginn eines Releaseprozesses zu beantworten, um festzustellen, ob ein neues Release ein sinnvolles Inkrement zu der vorherigen Version darstellen würde:

- Welche Änderungen gab es seit der letzten Version?
- Welche Plugins sollen dem Release hinzugefügt werden?
- Rechtfertigen die Änderungen ein MAJOR- oder MINOR-Release (s. 4.2.1 Produktlinien und Versionierung) bzw. überhaupt ein Release?
- Welche offenen Probleme bestehen aktuell und können die noch behoben werden? Gibt es insbesondere Dinge, die schlechter sind als vorher?
- Welche Änderungen in offenen Merge-Requests bzw. Branches sollen noch mit in das Release?

In den folgenden Abschnitten wird die Einordnung des Release in die Phasen der Softwareentwicklung und die Organisation des Releaseteams im universitären Lehrmodul beschrieben.

4.1.1 Einordnung des Release in die Phasen der Softwareentwicklung

Bei der Softwareentwicklung werden verschiedene Phasen durchlaufen. Traditionell lassen sich diese folgendermaßen grob beschreiben (Sommerville 2018, Kap.2):

1. Analyse und Definition der Anforderungen
2. System- und Softwareentwurf
3. Implementierung und Modultests
4. Integration und Systemtest
5. Betrieb und Wartung

Bevor also überhaupt ein Release bereitgestellt werden kann, müssen vorher bereits die anderen Schritte ausgeführt worden sein. Hierbei ist zu beachten, dass in den hier betrachteten universitären Projekten mit dem Einsatz von Scrum bzw. Scrum @ Scale eine agile, inkrementelle Entwicklung vorgenommen wird. Das bedeutet, dass diese Aktivitäten nicht zwingend nacheinander durchlaufen werden, sondern zum Teil gleichzeitig und iterativ ausgeführt werden.

Die Aufgaben eines Releaseteams in der RENEW-Entwicklung sind vor allem in den Phasen 4 und 5 anzusiedeln. Alle Aufgaben, die ein Releaseteam für die Veröffentlichung einer neuen Version von RENEW durchführen muss, sind im Kapitel „Erarbeitetes Vorgehen zur Durchführung eines Release“ festgehalten.

4.1.2 Organisation des Releaseteams im universitären Lehrmodul

An der Entwicklung der OSS RENEW sind viele Personen beteiligt. Die Entwicklung wird vornehmlich von Studierenden in universitärer Projektarbeit und innerhalb von Abschlussarbeiten vorgenommen. Die Projekte variieren in der Größe der Teilnehmendenzahlen, welche sich zwischen 20 und 60 bewegen (Anonymous Author(s) 2024). Die beteiligten Personen nehmen unterschiedliche Rollen bei der Entwicklung ein. Abschlussarbeitende haben

i.d.R. bereits Erfahrung in der RENEW-Entwicklung durch die Teilnahme an Lehrmodulen gesammelt.

In den Lehrmodulen werden den Studierenden in einer Onboarding-Phase die nötigen Grundlagen für die in der Entwicklung angewandten Methoden und Technologien vermittelt. Diese beinhalten insbesondere den agilen Softwareentwicklungs-Framework Scrum, die Software RENEW, die Versionsverwaltung mit Git bzw. GitLab, die Entwicklungsumgebung *IntelliJ IDEA*[1], das Ticketsystem *Jira*[2], die Wiki-Software *Confluence*[3], das Schreiben von Javadoc und von Unittests (Anonymous Author(s) 2024).

Danach werden die Teilnehmenden in themenbezogene Projektteams eingeteilt. Jedes Team bildet ein Scrum-Team, welches aus PO, SM und Entwickelnden besteht (s. 2.4.1 Rollen) und seine Arbeit in Sprints organisiert (s. 2.4.2 Ereignisse).

Im Lehrmodul wird die Arbeit der Teams untereinander durch die Anwendung des Frameworks Scrum @ Scale koordiniert (Anonymous Author(s) 2024). Eine ausführliche Beschreibung der Organisation der universitären Projekte lässt sich in "Tackling Project-Based Learning in a Large-Scale Educational Project: An Experience Report." finden (Anonymous Author(s) 2024).

Auch die Releaseteams, welche die Veröffentlichung von RENEW 2.6, 4.0 und 4.1 übernommen haben, waren als derartige Scrum-Teams organisiert. Innerhalb der Releaseteams wurden Aufgaben der Qualitätssicherung, des Testens und des Release Engineerings, sowie Bugfix- und DevOps-Tätigkeiten übernommen. Besonders wichtig war die Kommunikation innerhalb des jeweiligen Teams und nach außen zum CPO, zu Hauptentwickelnden und anderen Entwickelnden, da immer wieder Vorgehensweisen abgestimmt und die Arbeit evaluiert werden mussten, damit letztendlich ein Release für die Öffentlichkeit bereit war.

4.2 Versionierung und Aufbau von RENEWS Releases

In diesem Abschnitt werden als Erstes die Versionierung und Produktlinien von RENEW erklärt.

Es folgt eine Beschreibung davon, welche Komponenten von RENEW in welcher Form Bestandteil von Veröffentlichungen sind, im Unterabschnitt „Plugins, Distribution-Packages und veröffentlichte Artefakte".

Zuletzt werden die Lizenzbedingungen, unter denen die Software veröffentlicht wird, kurz wiedergegeben.

4.2.1 Produktlinien und Versionierung

Die Software RENEW wird in zwei übergeordneten Varianten entwickelt und bereitgestellt. Es gibt die klassische Variante, deren Versionen eine zwei an erster Stelle führen (2.x), und die modulare Variante, deren Versionen eine vier an erster Stelle führen (4.x). Beide Varianten haben jeweils eine aktuelle, veröffentlichte Version.

Innerhalb der Entwicklung von RENEW werden diese unterschiedlichen Varianten als unterschiedliche *Produktlinien* bezeichnet. Es gibt ein GitLab-Projekt mit dem Namen `Renew`[4], in welchem die Entwicklung von RENEW zentral verwaltet wird. Im GitLab-Projekt ist der Hauptbranch für die Entwicklung vom klassischen RENEW 2.x der `master`-Branch. Der Haupt-

[1] IntelliJ IDEA: https://www.jetbrains.com/idea/
[2] Atlassian Jira: https://www.atlassian.com/de/software/jira
[3] Atlassian Confluence: https://www.atlassian.com/de/software/confluence
[4] https://git.informatik.uni-hamburg.de/tgi/Renew

branch für das modulare RENEW 4.x war bis zum Oktober 2023 der `modular-master`-Branch, seitdem wurde auf den `main`-Branch gewechselt.

Die klassische Variante umfasst mehr Plugins als die modulare Variante (s. Anhang B), welche eine neue Benutzeroberfläche hat und durch den modularen Aufbau der Systemarchitektur leichter erweiterbar sein soll. Beide Produktlinien wurden bisher nebeneinander weiterhin gewartet. Im Wintersemester 2023/24 wurde die Entwicklung auf dem `master`-Branch jedoch größtenteils eingestellt.

In Abbildung 4.1 ist eine Übersicht zu RENEWs Produktlinien und den zugehörigen Versionen zu sehen. Die einzelnen Versionen sind jeweils einer Produktlinie – klassisch oder modular – zugeordnet. Die „2.x“ repräsentiert hierbei alle Vorgängerversionen von RENEW 2.6.

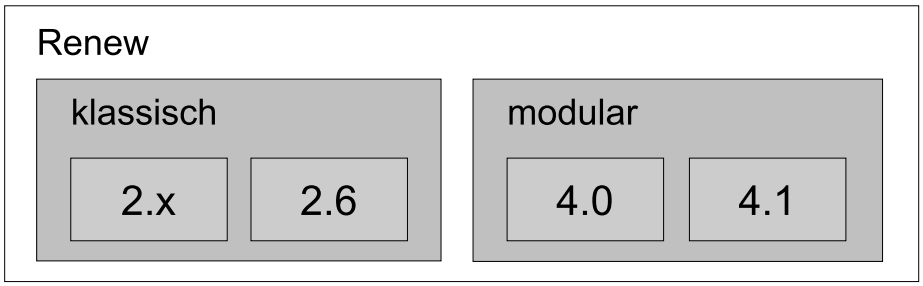


Abbildung 4.1: RENEWs Produktlinien und Versionen

Bei der *Versionierung*, also der Benennung der unterschiedlichen Versionen, wird weitestgehend der Definition von Semantic Versioning 2.0.0[5] entsprochen.

Das bedeutet, dass die Versionsnummern dem Schema MAJOR.MINOR.PATCH folgen. Die MAJOR-Versionsnummer wird erhöht, wenn mit vorhergehenden Versionen inkompatible Veränderungen vorgenommen wurden. Die MINOR-Versionsnummer wird erhöht, wenn rückwärtskompatible Funktionalitäten hinzugekommen sind. Die PATCH-Versionsnummer wird erhöht, wenn rückwärtskompatible Fehlerbehebungen vorgenommen wurden (Preston-Werner 2013).

Aus dem Kapitel „History“ des Handbuchs von RENEW lässt sich nachvollziehen, dass die jeweiligen Erhöhungen der Versionsnummern dieser Definition entsprechen (Kummer u. a. 2023b, S.30ff).

Der offensichtlich ungewöhnliche Sprung der MAJOR-Versionsnummer von 2 auf 4 ist damit zu erklären, dass ein RENEW mit der Versionsnummer 3.x zwar entwickelt, aber nie veröffentlicht und die Entwicklung eingestellt wurde.

Abweichend vom Schema des Semantic Versioning führen die meisten RENEW-Versionen keine dritte Stelle. Diese PATCH-Versionsnummer wurde nur in den Versionen explizit angegeben, wenn es sich bei ihnen tatsächlich um Maintenance-Releases handelte und die PATCH-Versionsnummer somit nicht 0 war. Implizit könnte hinter allen zweistelligen Versionsnummern der veröffentlichten RENEW-Versionen eine 0 gelesen werden.

Releases werden im GitLab-Projekt `Renew` nach einer erfolgreichen Veröffentlichung getaggt.

In Git sind Tags Pointer, die auf einen bestimmten Commit zeigen. Zusätzlich zum Tag-Namen können Tags noch weitere Informationen beinhalten, bspw. eine Tag-Nachricht. Tags werden häufig dazu verwendet, Releasezeitpunkte zu markieren (Chacon und Straub 2014, Kap.2).

[5] `https://semver.org/`

Die Release-Tags von RENEW haben die Form `renewMAJOR-MINOR-PATCH`, wobei die PATCH-Versionsnummer auch hier oft weggelassen wird. In Abbildung 4.2 sind die letzten drei Release-Tags von RENEW zu sehen, wie sie in der GUI von GitLab angezeigt werden[6].

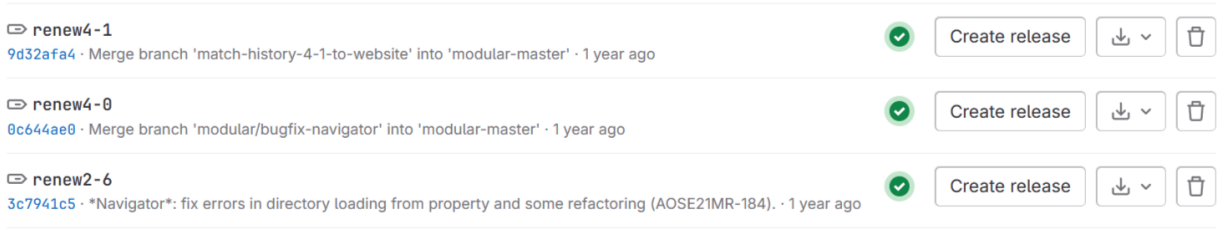


Abbildung 4.2: Release-Tags der Versionen 2.6, 4.0 und 4.1

Die Tags werden von Hauptentwicklern erstellt, sobald eine neue Version endgültig auf der Webseite veröffentlicht wurde.

Zusätzlich wird beim Erstellen der Release-Artefakte automatisch der SHA[7]-1-Hashwert des Snapshots, der für diesen Vorgang genutzt wurde, gespeichert und später automatisiert in einem Kommentar in der `download.html` der Version festgehalten. So ist auch ohne das Tag nachvollziehbar, welcher Commit der letzte vor der Veröffentlichung der Version war.

4.2.2 Plugins, Distribution-Packages und veröffentlichte Artefakte

Das modulare RENEW, welches auf dem `main`-Branch entwickelt wird, umfasst aktuell 65 Plugins; das klassische RENEW, welches auf dem `master`-Branch entwickelt wird, hat zur Zeit 77 Plugins. Eine Übersicht über die Plugins bzw. welche davon in den Releases 2.6, 4.0 und 4.1 veröffentlicht wurden ist in Anhang B zu finden.

Zu einer Veröffentlichung von RENEW gehören unterschiedliche kompilierte Pluginmengen, Teile des Sourcecodes des Projekts, ein Mac OS X Application Bundle, Installationsskripte und die Dokumentation in Form eines Handbuchs, mehrerer Readme-Dateien und einer Datei, in der die Lizenzbedingungen enthalten sind. Die Pluginmengen bilden zum einen unterschiedliche Distribution-Packages von RENEW[8] und zum anderen zwei verschiedene Ansammlungen zusätzlicher optionaler Plugins. Eine Auflistung der Pluginmengen vom Release 2.6 ist in Anhang B.1 und von den Releases 4.0 und 4.1 in Anhang B.2 zu finden.

Jede Veröffentlichung enthält ein sogenanntes Base-Package, welches die wichtigsten Plugins[9] für die Nutzung von RENEW als ausführbare JAR[10]-Dateien, benötigte Bibliotheken als JAR-Dateien, die Dokumentation, Beispielnetze, Installationsskripte und eine Kopie der Lizenzbedingungen beinhaltet.

Zusätzlich zu diesen Kernplugins werden die JAR-Dateien einiger optionaler Plugins und weiterer Extra-Plugins bereitgestellt, die mit dem Base-Package zusammen genutzt werden können. Die optionalen Plugins sind offizieller Teil einer Veröffentlichung. Die Extra-

[6] Alle Release-Tags von RENEW sind hier zu finden: `https://git.informatik.uni-hamburg.de/tgi/Renew/-/tags?search=renew&sort=updated_desc`

[7] **S**ecure **H**ash **A**lgorithm

[8] Diese Distribution-Packages werden auf der RENEW-Webseite nur „Packages“ genannt; in den Gradle-Dateien wird teils von „Distributionen“ geschrieben. Hier wird der Begriff „Distribution-Packages“ gewählt, um zu verdeutlichen, dass diese Packages nichts mit Java Packages bspw. zu tun haben.

[9] 15 Plugins in 2.6, 16 in 4.0 und 17 in 4.1

[10] **J**ava **Ar**chive

Plugins sind offiziell nicht veröffentlicht, auch wenn sie öffentlich herunterladbar sind, wenn die entsprechende, auf der Webseite nicht verlinkte URL bekannt ist.

Neben dem Base-Package gibt es weitere Distribution-Packages, welche die Nutzung von RENEW für bestimmte unterschiedliche Zwecke unterstützen sollen. So hat die Version 2.6 die Distribution-Packages Renew Analysis, Renew Plugin Development, Renew IDE und Renew CCPN. Renew Analysis erlaubt das Analysieren von Platz-/Transitions-Netzen, Renew Plugin Development unterstützt die Entwicklung zusätzlicher RENEW-Plugins, Renew IDE erweitert Renew Plugin Development, Renew CCPN enthält zusätzlich zum Base-Package einen Curry-Net-Compiler-Formalismus und die Funktionalität einen Reachability-Graph zu generieren. Die Versionen 4.0 und 4.1 haben nur ein zusätzliches Distribution-Package, welches Renew IDE heißt.

Außerdem hat jede Veröffentlichung ein Source-Package. Dieses enthält den Quelltext der in den Distribution-Packages veröffentlichten Plugins und der optionalen Plugins, die Gradle-Dateien, die für das Kompilieren benötigt werden, die JAR-Dateien der benötigten Bibliotheken, die Dateien `README`, `LICENSE` und `COPYING`, das Handbuch als PDF und die TEX-Dateien des Handbuchs.

Allerdings sind in den Source-Packages von 4.0 und 4.1 einige Plugins enthalten, die weder Teil des Base- noch des Renew IDE-Package und auch nicht der optionalen Plugins sind. Das rührt daher, dass diese in anderen (womöglich zukünftigen) Distribution-Packages Teil sein könnten – nämlich Renew Analysis und Renew CCPN – analog zu den Distribution-Packages von 2.6. Es existieren bereits die jeweiligen Gradle-Tasks für diese beiden Distribution-Packages, welche die Base-Distribution um das Momoc- und das MomocGui-Plugin bzw. das CCPN- und das RGBase-Plugin erweitern, jedoch wurden sie in den bisherigen modularen Veröffentlichungen nicht inkludiert.

Das Mac OS X Application Bundle enthält neben den Plugins des Base-Packages auch das AppleUI-Plugin.

Für RENEW existieren Installationsskripte, die die Installation der Software unter Unix- und Windowssystemen erleichtern sollen. Sie sind im `Core/bin`-Ordner des GitLab-Projekts `Renew` zu finden.

Das Handbuch ist eine PDF-Datei, welche aus TEX-Dateien generiert wurde. Die entsprechenden Dateien sind im `Core/doc`-Ordner des GitLab-Projekts `Renew` zu finden. Die Readme-Dateien sind im `Core/doc`- bzw. im `Core/mac`-Ordner hinterlegt.

Die Lizenzbedingungen werden in einer TEX-Datei des Handbuchs festgehalten, aus welcher später zusätzlich zu einem entsprechenden Abschnitt im Handbuch eine Datei namens `LICENSE` generiert wird.

4.2.3 RENEWS Lizenzbedingungen

Die veröffentlichten Versionen von RENEW sind unentgeltlich erhältlich, aber nicht ohne Einschränkungen. Der Großteil der Software ist unter der GNU[11] Lesser General Public License, Version 2.1 (LGPL-2.1)[12] veröffentlicht (Kummer u. a. 2023b). Die LGPL-2.1 ist konform mit der OSD (Feller und Fitzgerald 2002, S.17). Die aktuellste Version der LGPL ist die Version 3.0 [13].

RENEW enthält anders lizenzierte Software externer Parteien. Diese sind unter folgenden Lizenzen veröffentlicht: Apache License, Version 2.0; 2-Clause BSD License; 3-Clause

[11] **G**NU's **N**ot **U**nix

[12] GNU Lesser General Public License, Version 2.1, `https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`

[13] GNU Lesser General Public License, Version 3.0, `https://www.gnu.org/licenses/lgpl-3.0`

BSD License; GNU LGPL; eigene Lizenzen (Kummer u. a. 2023b). Die ersten vier sind alle konform mit der OSD (The Open Source Initiative 2023a).

Insgesamt zielt die Lizenz von RENEW darauf ab, mit der OSD der OSI konform zu sein. Bei gefundenen Verstößen bitten die Autoren des Nutzerhandbuchs darum, sie darauf hinzuweisen (Kummer u. a. 2023b).

Die Lizenzbedingungen sind sowohl in RENEWs Handbuch, sowie in einer `LICENSE`-Datei, die in den Distribution-Packages enthalten ist, als auch auf der Webseite `www.renew.de` zu finden.

4.3 Branching und Qualitätssicherung bei Änderungen auf dem Hauptbranch

Im GitLab-Projekt `Renew` existieren zwei Hauptbranches für RENEWs Entwicklung. Für die Entwicklung der klassischen Version ist es der `master`-Branch. Für die Entwicklung der modularen Version wird seit Oktober 2023 der `main`-Branch als Hauptbranch verwendet.

Auf den Hauptbranches liegt der jeweils aktuelle Stand der beiden Versionen. Änderungen, die hier integriert werden, haben Qualitätssicherungsmaßnahmen durchlaufen und sollten für eine Veröffentlichung bereit sein.

Im Folgenden wird erläutert, welche Branching-Konventionen existieren und wie Änderungen nach der Durchführung von Qualitätssicherungsmaßnahmen in den Hauptbranch `main` übernommen werden. Diese Erläuterungen gelten für die Entwicklung vom modularen RENEW. Das klassische RENEW wurde zum einen von dieser Arbeit nicht in der Entwicklung, sondern nur bei der Veröffentlichung von Version 2.6 begleitet, zum anderen wurde die Entwicklung auf dem `master`-Branch im Wintersemester 2023/24 größtenteils eingestellt.

4.3.1 Branching-Konventionen

RENEW wird von Studierenden innerhalb von Abschlussarbeiten und Projekt- bzw. Praktikumsmodulen weiterentwickelt. Die Arbeit in den Lehrprojekten wird in Teams aufgeteilt, die sich jeweils unterschiedlichen Themengebieten bzw. Entwicklungsfeldern annehmen.

Abbildung 4.3 gibt eine vereinfachte Übersicht darüber, wie die Branch-Struktur im GitLab-Projekt `Renew` laut Konvention aussehen soll.

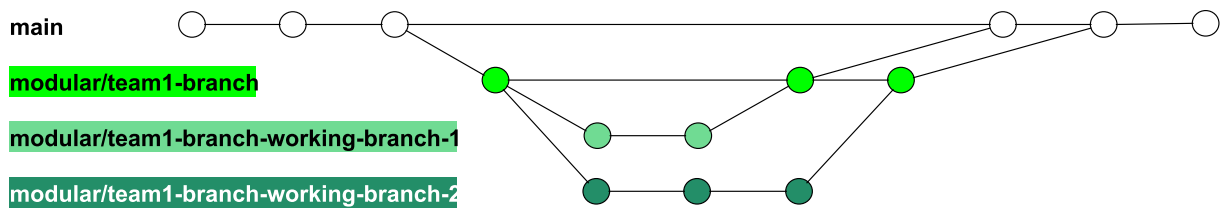


Abbildung 4.3: Branching-Struktur laut Konvention

Zu Beginn des Lehrprojekts wird ein Teambranch von jedem Team angelegt, welcher vom Hauptbranch `main` abgezweigt wird. Analog legen Abschlussarbeitende einen eigenen Branch an. Der Name dieses Branches beginnt mit dem Präfix `modular/`, dem ein prägnanter Ausdruck aus Kleinbuchstaben und Bindestrichen folgt, der das Themengebiet des Teams bzw. der/des Abschlussarbeitenden reflektiert. In Abbildung 4.3 heißt dieser Teambranch `modular/team1-branch`.

Wenn eine entwickelnde Person etwas an der Codebasis ändern möchte – sei es um einen Bugfix vorzunehmen, Dokumentation oder Tests zu ergänzen, eine neue Funktionalität hinzuzufügen oder sonstiges –, so passiert dies auf einem Arbeitsbranch. Der Arbeitsbranch wird vom Teambranch abgezweigt und erhält einen Namen, der sich aus dem Namen des Teambranches und der Bezeichnung der vorzunehmenden Änderung in Kleinbuchstaben und Bindestrichen zusammensetzt. In Abbildung 4.3 heißen diese Branches `modular/team1-branch-working-branch-1` und `modular/team1-branch-working-branch-2`.

Die Konventionen für das Branching im RENEW-Projekt sind auf einer Confluence-Seite festgehalten, die für an der Entwicklung beteiligte Personen zugänglich ist. Dort ist auch ersichtlich, wie Commit-Nachrichten aufgebaut sein müssen, um später den Ansprüchen der Qualitätssicherung zu genügen (s. Renew-Development-Team 2023b).

4.3.2 Qualitätssicherung durch zweistufigen Reviewprozess

Nachdem Entwicklungen auf einem Arbeitsbranch vorgenommen wurden und diese von der entwickelnden Person als bereit für die Überführung in den Hauptbranch angesehen werden, durchlaufen sie einen zweistufigen Reviewprozess bevor sie durch Merging in die Codebasis übernommen werden. Dies geschieht indem zunächst ein Merge Request (MR) in GitLab für den Arbeitsbranch gegen den Teambranch und später ein MR für den Teambranch gegen den Hauptbranch erstellt wird.

In Abbildung 4.4 ist eine workflownetzartige Darstellung des zweistufigen Reviewprozesses dargestellt[14].

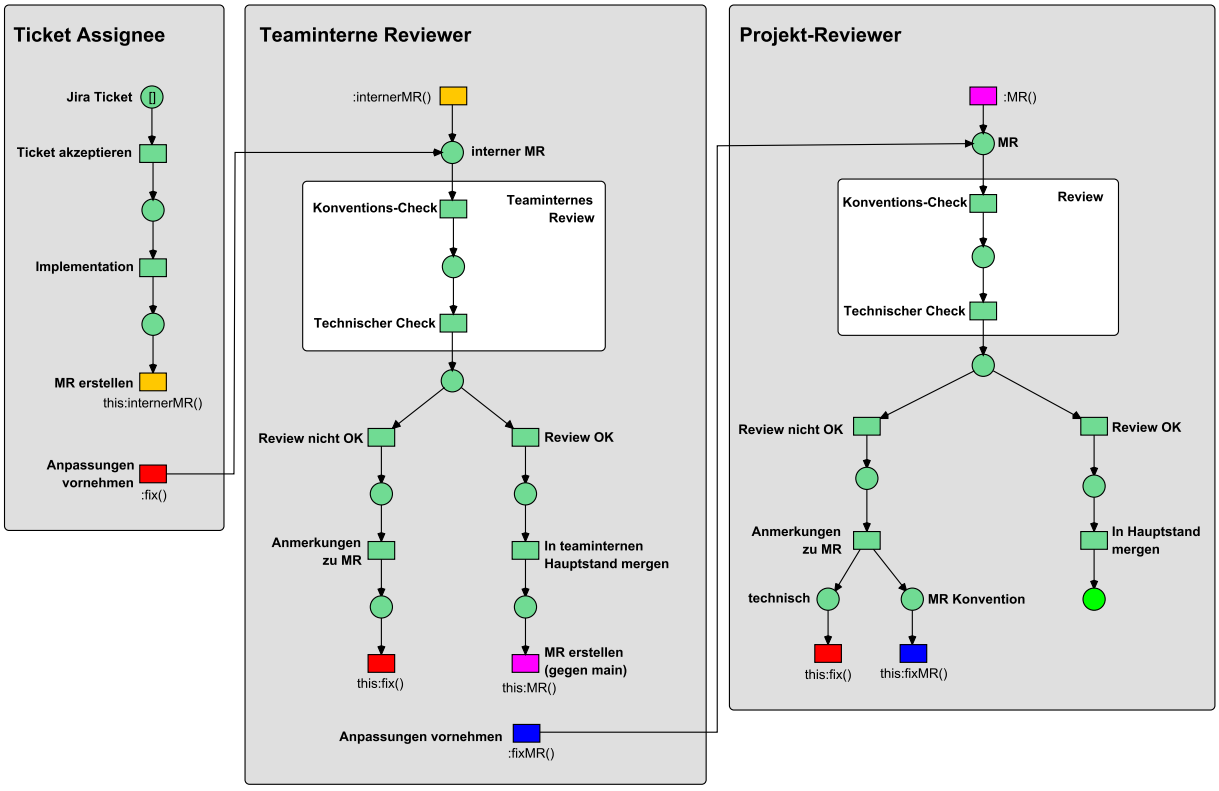


Abbildung 4.4: Merge Requests und zweistufiger Reviewprozess

[14] Es handelt sich dabei um eine Adaption des von Clasen und Hansson erstellten Workflownetzes (Clasen 2023). Bei der Adaption wurden durch die Nutzung von synchronen Kanälen die verschiedenen am Review beteiligten Rollen getrennt, wobei synchrone Kanäle durch den Einsatz von korrespondierenden Farben gekennzeichnet wurden.

Zunächst sind die Aufgaben des Ticket Assignees, also der entwickelnden Person, die eine in einem Jira-Ticket festgehaltene Aufgabe bearbeitet, dargestellt. Diese sind: Ticket akzeptieren, Implementation, MR erstellen und evtl. Anpassungen vornehmen.

Wurde ein MR durch den Ticket Assignee gegen den Teambranch erstellt, können teaminterne Reviewer tätig werden. Innerhalb des teaminternen Reviews erfolgen eine Überprüfung der Einhaltung der Konventionen und eine technische Überprüfung der Funktionalität, sofern dies nötig ist – bei Ergänzung von fehlender Dokumentation ist dies bspw. nicht der Fall. Falls innerhalb dieses Reviews Mängel feststellt werden, werden Anmerkungen im MR festgehalten, die vom ursprünglichen Ticket Assignee bearbeitet werden müssen. Nach der Überarbeitung kann das Review erneut durchgeführt werden, wobei bereits überprüfte, nicht bemängelte Dinge nicht doppelt geprüft werden. Falls ein MR freigegeben wurde, werden die Änderungen in den Teambranch aufgenommen und der MR geschlossen.

Spätestens am Ende eines Lehrprojekts bzw. einer Abschlussarbeit wird ein MR für den Teambranch bzw. den Branch der/des Abschlussarbeitenden gegen den `main`-Branch geöffnet. Das Review erfolgt an dieser Stelle von Projekt-Reviewern, die bereits viel Erfahrung in der Entwicklung von RENEW haben; der Ablauf ist derselbe wie beim teaminternen Review.

Für den Reviewprozess existiert eine *Code Review Checkliste* auf einer Confluence-Seite, die für an der Entwicklung beteiligte Personen zugänglich ist (s. Renew-Development-Team 2024). Die Checkliste erläutert sowohl die Vorgehensweise als auch die zu überprüfenden Punkte, welche mit den Konventionen für die Entwicklung von RENEW übereinstimmen müssen. Geprüft wird die Einhaltung der Merge-Request- und der Jira-Ticket-Konventionen, die Erfüllung der Kriterien, die an die Funktion gestellt werden, und die Einhaltung der Konventionen für Git, Java, Code und Dokumentation. Die Checkliste ist für beide Reviews dieselbe und ist in Anhang C zu finden.

4.4 Buildmanagement und CI/CD von RENEW

Die Verwendung von Buildmanagement-Tools und CI/CD-Pipelines unterstützt sowohl die Entwicklung als auch die Veröffentlichung von RENEW.

In diesem Abschnitt wird zunächst auf RENEWs Buildmanagement für das Release eingegangen, für welches bei 2.x-Versionen *Apache Ant* und bei 4.x-Versionen *Gradle* genutzt wird.

Darauf folgen Beschreibungen der Umsetzung der CI/CD-Pipelines, wobei auf die Runner, die Stages und Jobs auf den Branches der unterschiedlichen Versionen und auf einige Veränderungen in den YAML-Dateien seit der letzten Veröffentlichung eingegangen wird.

4.4.1 RENEWs Buildmanagement für das Release

Buildmanagement-Tools erlauben es, den Bauprozess einer Software zu automatisieren und diesen dabei durch die explizite Konfiguration der Build-Umgebung konsistent und einheitlich zu gestalten. Außerdem können diese Werkzeuge sich um die Abhängigkeiten der Software kümmern, indem sie diese herunterladen und cachen (Heinze 2022).

Innerhalb der RENEW-Entwicklung werden die Buildmanagement-Tools Apache Ant und Gradle verwendet.

Im Folgenden werden die Gradle-Release-Tasks für 4.x-Versionen und die Ant-Release-Targets für 2.x-Versionen von RENEW beschrieben.

Gradle-Release-Tasks für 4.x

RENEWs Pakete und sonstige Dateien, die für die Veröffentlichung einer modularen Version benötigt werden, werden mit Hilfe von *Gradle*[15] gebaut. Gradle ist ein Buildmanagement-Werkzeug. Das bedeutet, dass mit Hilfe von Gradle Java-Klassen kompiliert und fertige Programme erzeugt werden können (Hansson 2020). Gradle bietet außerdem ein umfangreiches Anhängigkeitsmanagement an (Heinze 2022).

Im GitLab-Projekt `Renew` liegen auf dem `main`-Branch auf oberster Ordnerebene folgende Dateien, welche die Verwendung von Gradle steuern:

- Der Ordner `gradle/wrapper` und die Skripte `gradlew` und `gradlew.bat` gehören zum Gradle-Wrapper, durch den die Gradle-Version für das Projekt standardisiert spezifiziert und bereitgestellt wird (Gradle Inc. 2023b).
- Die `settings.gradle`-Datei ist eine Konfigurationsdatei, in der die Namen des Hauptprojekts (RENEW) und seiner Unterprojekte (RENEWs Plugins) spezifiziert werden.
- Die `gradle.properties`-Datei wird genutzt um Eigenschaften global für das Projekt zu definieren.
- Die `build.gradle`-Datei ist das Build-Skript. Hier wird der Bauprozess konfiguriert und die Gradle-Tasks werden dafür spezifiziert.
- In der `plugin-names.gradle`-Datei werden verschiedene Stringarrays in Variablen gespeichert und für das Projekt global verfügbar gemacht, welche in den Dateien `build.gradle` und `release-tasks.gradle` genutzt werden. Sie stellen verschiedene Sets aus der Menge aller RENEW-Plugins dar.
- Die `release-tasks.gradle`-Datei enthält die Gradle-Tasks, die für die Erzeugung der Dateien für eine Veröffentlichung auf der Webseite benötigt werden.
- Die `release-packages.gradle`-Datei enthält Task-Generatoren, welche für die Erstellung der unterschiedlichen Distribution-Packages (Base, Analysis, IDE, CCPN) in der `release-tasks.gradle` genutzt werden.

Zusätzlich zu diesen Gradle-Dateien besitzt jedes RENEW-Plugin jeweils seine eigene `build.gradle`- und `gradle.properties`-Datei. In der `gradle.properties` wird der Modulname des Plugins festgehalten, in der `build.gradle` werden Abhängigkeiten deklariert (Hansson 2020).

Für die Veröffentlichung einer neuen modularen Version von RENEW wird die Gradle-Task `release` vom GitLab CI/CD-Job `release:packages` in der `release`-Stage aufgerufen (s. Stages und Jobs auf dem `modular-master`). Innerhalb der `release-tasks.gradle` wird die `release`-Task definiert. Vor dem AOSE22-Projekt[16] bzw. vor der Veröffentlichung von RENEW 4.1 wurde er noch in der `build.gradle` definiert[17]. Die Auslagerung in eine eigene Datei soll zum einen der Übersichtlichkeit dienen, zum anderen wird die `build.gradle` zusammen mit dem Quelltext von RENEW veröffentlicht, wobei sie aber nicht die für die Veröffentlichung relevanten Tasks enthalten soll.

Am Anfang der `release-tasks.gradle` wird die Versionsnummer der zu veröffentlichenden Version von RENEW spezifiziert. Ganz am Ende wird der `release`-Task definiert. Dieser ist abhängig von anderen vorher definierten Gradle-Tasks, sodass diese zuerst von Gradle ausgeführt werden, wenn `./gradlew release` aufgerufen wird.

[15] https://gradle.org/

[16] Die AOSE-Projekte und -Praktika sind die Lehrmodule, in denen RENEW weiterentwickelt wird. AOSE steht für **A**gent **O**riented **S**oftware **E**ngineering.

[17] Dokumentation der Anpassung: https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?spaceKey=AOSE22QLTY&title=Improvements+to+the+build.gradle+file

In diesen wird alles gebaut und in die richtigen Ordner kopiert: die Distribution-Packages „Base“, „Analysis“, „IDE“, „CCPN“ und „Source“, die optional herunterladbaren RENEW-Plugins und weitere zusätzliche RENEW-Plugins, die nicht explizit auf der Webseite zum Download bereitgestellt werden, die Lizenzdatei, das Handbuch, Installationsskripte, die Apple-Distribution und Beispieldateien.

Für die verschiedenen Distribution-Packages werden in der `plugin-names.gradle`-Datei die jeweils benötigten Plugins in Stringarray-Variablen gespeichert. Die „Base“-Distribution enthält (fast) alle Plugins aus `plugin_names_dist`, „Analysis“ alle aus dem Stringarray `plugin_names_analysis`, „IDE“alle aus `plugin_names_ide` und „CCPN“ alle aus dem Stringarray `plugin_names_ccpn`. Die letzteren drei Stringarrays enthalten jeweils alle Namen der Plugins aus `plugin_names_dist` und unterschiedliche weitere Pluginnamen.

Die Plugins „Refactoring“ und „RenewAnt“ werden zwar im `plugin_names_dist`-Stringarray aufgeführt, allerdings sind sie nicht in den Distribution-Packages enthalten. Der Grund dafür ist, dass sie in der `release-packages.gradle` wieder exkludiert werden.

„Source“ enthält alle Plugins aus `plugin_names_all` und noch weitere, welche in der `release-tasks.gradle` angegeben werden. Zu beachten ist, dass Plugins, die nicht auf Gradle umgestellt wurden, nicht gebaut werden und somit später auch nicht in den generierten Distribution-Packages zu finden sind.

Ant-Release-Targets für 2.x

Für nicht-modulare Versionen von RENEW, also klassische Versionen mit Nummern 2.x, wird *Apache Ant*[18] als Werkzeug für das Management und die Automatisierung des Bauprozesses verwendet.

Die Ant-Targets, die vom GitLab CI/CD-Job `packages` in der `packages`-Stage aufgerufen werden (s. Stages und Jobs auf dem `master`), sind in der `build-dist.xml` auf dem `master`-Branch festgehalten. Sie sind analog zu den Release-Tasks der modularen Version, wobei sich die Pluginmengen der Distributionen zum Teil unterscheiden.

Auch der Wert für das Property `renew-version`, also die Versionsnummer, wird in der `build-dist.xml` gesetzt.

4.4.2 Umsetzung der CI/CD-Pipelines

Innerhalb des GitLab-Projekts `Renew` existieren unterschiedliche YAML-Dateien, mit denen die CI/CD-Pipelines des Projekts konfiguriert werden.

Im Folgenden sollen sowohl die Implementation von GitLab Runner, sowie der Aufbau und die Funktion der `.gitlab-ci.yml` für den `master`-Branch, auf dem RENEW in der klassischen Version liegt, und für den `modular-master`-Branch, auf dem RENEW in der modularen Version liegt[19], als auch Veränderungen in den YAML-Dateien seit der letzten Veröffentlichung, erläutert werden.

Runner

Der Runner, welcher für RENEW verwendet wird, ist als Group Runner auf dem Server des Rechenzentrums registriert. Die tgi[20]-Gruppe hat dort zwei Runner (s. Abbildung 4.5), näm-

[18] `https://ant.apache.org/`

[19] Seit Oktober 2023 hat der `main`-Branch den `modular-master`-Branch als Hauptbranch für die Entwicklung des modularen RENEW abgelöst.

[20] TGI steht für den Arbeitsbereich Theoretische Grundlagen der Informatik, der inzwischen Algorithmen, Randomisierung und Theorie (ART) heißt.

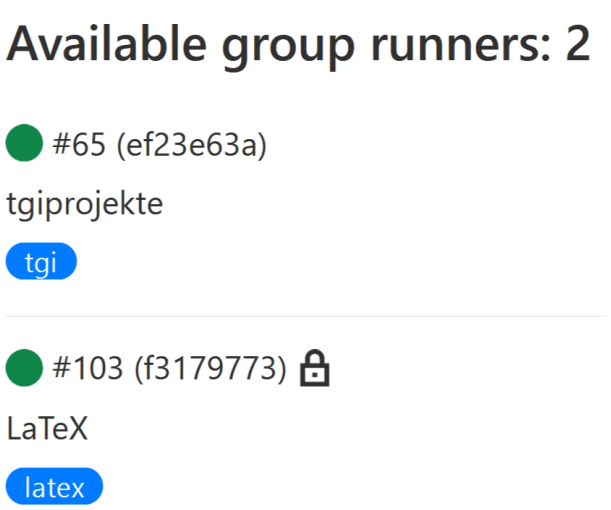


Abbildung 4.5: Group Runner der tgi-Gruppe

lich den Runner `tgiprojekte` mit dem Tag `tgi` und den LaTeX-Runner mit Tag `latex`, welcher u.a. für die Git-Repositories, in denen Abschlussarbeiten aus dem TGI-Bereich liegen, genutzt wird. `tgiprojekte` wird für die hier beleuchteten Versionen von RENEW verwendet.

Die zugehörigen Docker Images sind in der Container Registry von GitLab CI/CD zu finden. In diesen Containern werden die in den YAML-Dateien spezifizierten Befehle ausgeführt.

Stages und Jobs auf dem `master`

Die Stages auf dem `master`-Branch werden in der dortigen `.gitlab-ci.yml`-Datei festgelegt und sind:

- `build:dist`, `build:all`, `build:plugins`, in diesen wird RENEW mit seinen Plugins gebaut,
- `deploy` baut die ausführbaren Dateien für MacOS und Windows, außerdem werden Docker Images gebaut und in der Container Registry von GitLab CI/CD gespeichert,
- `deploy:plugins` triggert *Mulans* Pipeline als Downstream-Pipeline, da es ein von RENEW abhängiges Projekt, ist und
- `packages` baut RENEWs Distribution-Packages bzw. alle für eine Veröffentlichung relevanten Dateien.

Die genaue Beschreibung der Stages und Jobs lässt sich in Anhang C.2 der Bachelorarbeit von Feldmann finden (s. Feldmann 2019). Seit der Erstellung seiner Arbeit hat sich Folgendes verändert:

- Allen Jobs wurde der Tag `tgi` hinzugefügt, der spezifiziert, dass der `tgi`-Runner verwendet werden soll.
- Des Weiteren wird nun in allen Jobs, in denen nicht Docker Images gebaut werden, das Docker Image mit dem Tag `latest` angegeben.
- In der `build:plugins`-Stage sind die Jobs `build:distribute` und `build:rmtdeps` hinzugekommen.

- Docker Images werden nicht länger mit Docker in Docker gebaut sondern mit kaniko[21].
- In der `deploy`-Stage sind die Jobs `deploy:docker:distribute` und `deploy:docker:-graphical` hinzugekommen.
- Einige Dependenzen von Jobs haben sich geändert bzw. sind erst jetzt explizit aufgeführt.
- Die `packages`-Stage ist hinzugekommen. Sie enthält den gleichnamigen Job `packages`, der manuell getriggert werden muss und dann alle für einen Release benötigten Dateien generiert und als Artefakt speichert.

Stages und Jobs auf dem `modular-master`

Zwischen den Veröffentlichungen von RENEW 4.0 und 4.1 hat sich die `.gitlab-ci.yml`-Datei auf dem `modular-master`-Branch nicht verändert, weswegen die darin enthaltenen Stages und Jobs für beide Versionen zusammengefasst beschrieben werden. Die Stages auf dem `modular-master`-Branch heißen `build`, `test`, `deploy` und `release`.

Die Jobs in den Stages `build` und `test` werden immer automatisch ausgeführt.
In der `build`-Stage sind die Jobs `build:all`, `build:dist` und `build:mulan`. In ihnen wird RENEW in der jeweiligen Konfiguration mit Gradle gebaut und nach erfolgreichem Bauen als Artefakt gespeichert.
In der `test`-Stage sind die Jobs `test:all` und `test:dist`. In diesen Jobs werden alle vorhandenen Tests für jeweils alle Module bzw. alle, die in `pluhin_names_dist`(s. Gradle-Release-Tasks für 4.x) sind, mit Gradle ausgeführt[22].

Jobs in der `deploy`-Stage werden nur ausgeführt, wenn die Pipeline durch einen Push auf den `modular-master` oder auf `modular/ci-update` ausgelöst wurde.
Die Jobs heißen `deploy:docker` und `deploy:docker:builder`. In `deploy:docker` wird ein Docker Image gebaut, welches genutzt werden kann, um RENEW und Mulan laufen zu lassen, während in `deploy:docker:builder` ein Docker Image gebaut wird, mit Hilfe dessen RENEW und Mulan gebaut werden können.
Die Images werden mit dem Tag `modular` bzw. `builder-modular` versehen und in der Container Registry der GitLab CI/CD gespeichert; sie werden allerdings zur Zeit nicht im Projekt verwendet[23].

Die Stage `release` hat den Job `release:packages`, dieser wird aber manuell getriggert, was mit `when: manual` spezifiziert wird. Die erzeugten Dateien werden nicht weiter ausgeliefert, sondern der Job macht sie nach erfolgreicher Ausführung – spezifiziert durch `when: on_success` – lediglich als Artefakt `packages.zip` verfügbar. Der Befehl, welcher für die Erzeugung aufgerufen wird, lautet

```
xvfb-run -a ./gradlew clean release --build-cache --stacktrace
```

`xvfb-run -a` ist ein Wrapper-Befehl, der es ermöglicht, die folgenden Befehle in einer virtuellen X-Serverumgebung auszuführen (Ubuntu Manpage Repository 2019).
`./gradlew` wird aufgerufen um die darauf folgenden Build-Tasks mit dem Gradle-Wrapper auszuführen (Gradle Inc. 2023b).
Der Aufruf der `clean`-Task stellt sicher, dass der Ordner `dist`, welcher als `destination` für

[21] https://cloud.google.com/blog/products/containers-kubernetes/introducing-kaniko-build-container-images-in-kubernetes-and-google-container-builder-even-without-root-access

[22] Hierbei werden allerdings einige Module ausgelassen, da deren Tests nicht erfolgreich ausgeführt werden.

[23] In den Stages `build` und `test` wird das Image `gradle:jdk11` verwendet, in der `build`-Stage wird das Image `gcr.io/kaniko-project/executor:debug` verwendet und in der `release`-Stage wird ein Image mit dem Tag `latest` verwendet.

Builds in der `gradle.properties`-Datei angegeben ist, mit allen darin enthaltenen Dateien gelöscht wird, bevor die nächste Task aufgerufen wird.
`release --build-cache --stacktrace` ruft die Gradle-Task `release` auf.
Die beiden angehängten Argumente bedeuten, dass Gradle versuchen wird, gecachten Output aus vorherigen Builds zu verwenden, und dass der Stacktrace auf der Kommandozeile ausgegeben wird (Gradle Inc. 2023a).
Die `release`-Task selbst wird in der `release-tasks.gradle`-Datei spezifiziert (s. Gradle-Release-Tasks für 4.x).

Veränderungen in den YAML-Dateien seit 4.1

Seitdem der Defaultbranch des RENEW-Projekts für die Entwicklung des modularen RENEW (4.x) der `main`-Branch ist, wurden die jeweiligen Referenzen von `modular-master` zu `main` in der YAML-Datei geändert.

Die Java-LTS[24]-Version, die für RENEW verwendet wird, wurde von 11 auf 17 geändert, weswegen auch die verwendeten Docker-Images in der `gitlab-ci.yml` angepasst wurden.

Es wurde neu festgelegt, wann Pipelines gestartet werden. Mit dem `workflow`-Schlüsselwort wurde allgemein angegeben, dass eine Branch-Pipeline gestartet wird, wenn es keinen offenen Merge-Request für diesen Branch gibt, und dass eine Merge-Request-Pipeline gestartet wird, wenn es einen offenen Merge-Request für den jeweiligen Branch gibt. Der Job `packages:release` hat eine spezielle Regel für die Ausführung. Zuvor wurde der Job nur manuell gestartet. Nun wird er automatisch ausgeführt, wenn der Zielbranch, in den gemergt werden soll, der Defaultbranch `main` ist. Bei anderen Zielbranches muss der Job weiterhin manuell gestartet werden, wenn er ausgeführt werden soll.

Die `test`-Stage wurde auf die zwei Stages `unit test` und `integration test` aufgeteilt, welches die Überlegungen zum automatisierten Testen aus der Abschlussarbeit von Al Shriteh reflektiert. `unit test` hat den Job `unit:test` und `integration test` hat den Job `integration:test`. Außerdem wurde der Cache aktiviert, um die benötigte Zeit für die Ausführung der Jobs zu verringern (Al Shriteh 2022). Des Weiteren werden die Testergebnisse als Artefakt gespeichert (Al Shriteh 2022).

[24] **L**ong **T**erm **S**upport

Kapitel 5

Erarbeitetes Vorgehen zur Durchführung eines Release

Dieses Kapitel erläutert, wie neue Versionen von RENEW für die Veröffentlichung vorbereitet und auf der Webseite `http://www.renew.de/` veröffentlicht werden, wodurch ergänzend zu Kapitel 4 die Einordnung, wie ein Veröffentlichungsprozess für ein umfangreiches, langjähriges, universitäres OSS-Projekt gestaltet werden kann, geleistet wird.

Dafür werden alle Tätigkeiten beschrieben, die nötig sind, um eine neue Version von RENEW zu veröffentlichen. Da in dieser Arbeit die Releases von RENEW 2.6, 4.0 und 4.1 betrachtet werden, wird auf die Veröffentlichung dieser Versionen Bezug genommen. Manche Arbeitsschritte wurden erst nach der Durchführung der Releases als nötig identifiziert und sind dennoch hier aufgenommen worden.

Die in diesem Kapitel beschriebenen Vorgehensweisen folgen auf die Herstellung eines releasefähigen Zustandes, was beinhaltet, dass alle neu zu veröffentlichenden Funktionen fertig implementiert wurden und stabil laufen und dass alle bisher gefundenen Bugs behoben oder als „Known Issues“ dokumentiert wurden.

Die Herstellung dieses Zustandes ist Teil der regulären Entwicklung, jedoch können während der Vorbereitung einer Veröffentlichung durch das Releaseteam weitere Programmfehler auffallen, die dann behoben werden müssen. Das Beheben eventueller Bugs ist jeweils individuell, kann deswegen schlecht generalisiert dokumentiert werden und wird aus diesem Grund hier nicht weiter beschrieben. Das generelle Vorgehen für das Integrieren von Änderungen wurde bereits innerhalb des vorhergehenden Kapitels beschrieben (s. 4.3 Branching und Qualitätssicherung bei Änderungen auf dem Hauptbranch).

Im Folgenden wird zuerst die genaue Durchführung der zur Vorbereitung eines Releases gehörenden Schritte „Erstellung des Changelogs“, „Überprüfung und Erhöhung von Versionsnummern“ und „Aktualisierung der Dokumentation“ innerhalb der RENEW-Entwicklung erklärt.

Es folgt eine Beschreibung der für die Veröffentlichung nötigen Vorbereitung der Webseiten; nämlich die Erstellung eines Prepare-Branches, die Generierung eines Versionsordners und die händischen Anpassungen.

Danach wird die Prüfung und der Review der Vorbereitungen im Abschnitt „Zweistufige Qualitätssicherung“ geschildert.

Zuletzt wird geschildert, auf welchen Server und mit welchem Zugriff das Deployment des Release passiert.

Um eine grobe Übersicht über die hier beschriebenen Arbeitsschritte zu geben, sind diese in Abbildung 5.1 in einem Workflownetz dargestellt zu finden.

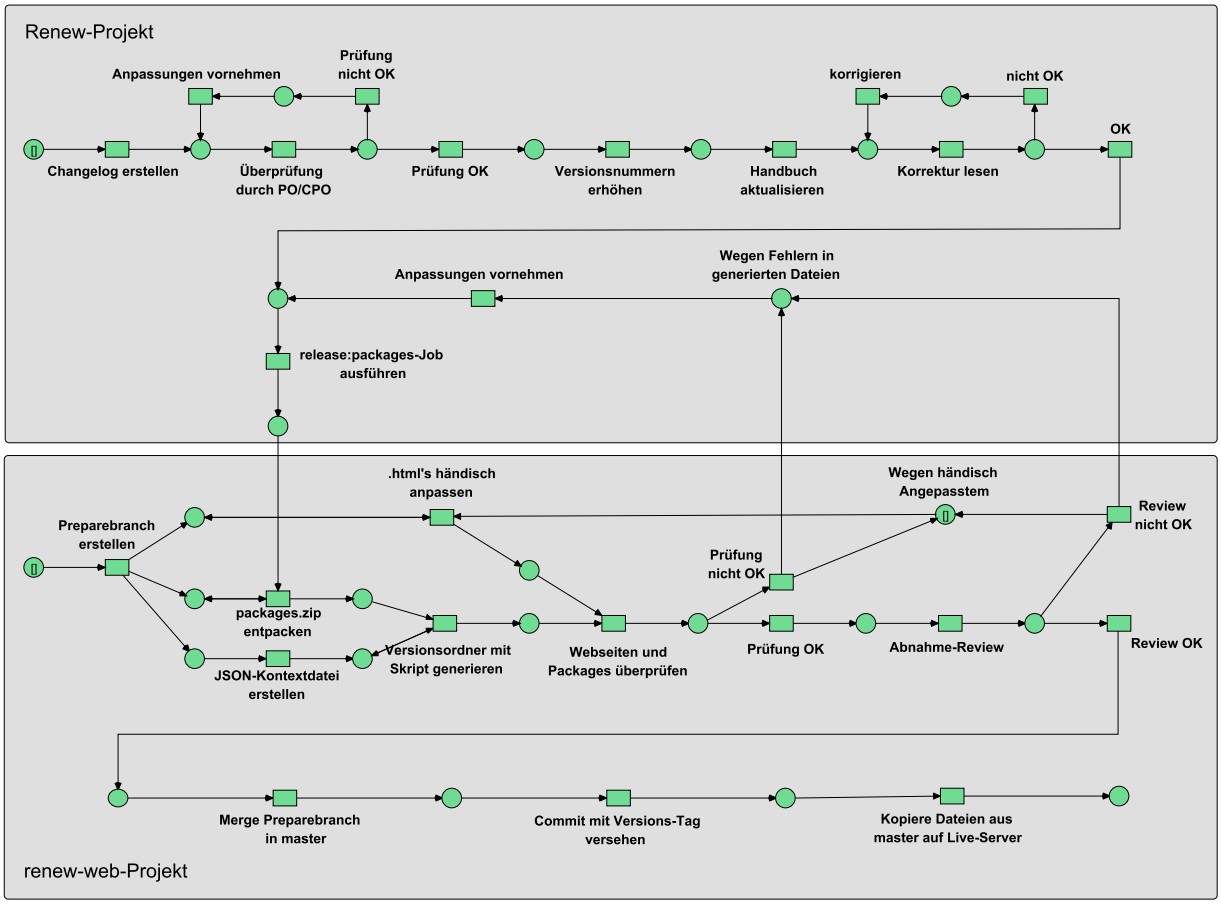


Abbildung 5.1: Durchführung eines Release

5.1 Erstellung des Changelogs

Um die Veröffentlichung von bereits entwickelten Neuerungen vorzubereiten, muss zunächst ein Changelog erstellt werden. In diesem werden die relevantesten Neuerungen zusammengefasst, die durch die zu veröffentlichende Version eingeführt werden.

Die Veränderungen werden sowohl im Handbuch als auch auf der Webseite von RENEW unter der Überschrift „History“ bzw. „Release History“ und den Unterüberschriften „Additions“ (dt.: Ergänzungen), „Removals“ (dt.: Entfernungen), „Modifications“ (dt.: Abwandlungen) und „Relevant for Developers“ aufgeführt. Die ersten drei Kategorien beinhalten Informationen zu Veränderungen, die für alle Nutzenden der gebauten Software relevant sind, die letzte enthält Informationen für Entwickelnde, die das Source-Package nutzen wollen, um es eigenständig zu verändern oder zu erweitern.

Um diese Veränderungen zusammenzutragen, müssen alle Commits gesichtet werden, die seit der letzten Veröffentlichung einer RENEW-Version gemacht wurden. Der letzte auf dem Hauptbranch vor der Generierung der veröffentlichten Artefakte getätigte Commit wird nach der erfolgreichen Veröffentlichung einer neuen Version mit einem Tag der Form `renew[MAJOR]-[MINOR]-[PATCH]`[1] versehen. Die Neuerungen können mit einem einfachen Vergleich[2] zwischen den Revisionen, auf die das Tag des letzten Release und der Hauptbranch zeigen, angezeigt werden. Für klassische 2.x-Versionen passiert dies auf dem `master`-

[1] Bspw. `renew4-0` oder `renew2-5-1`, s. `https://git.informatik.uni-hamburg.de/tgi/Renew/-/tags?search=renew&sort=updated_desc`

[2] Vergleich der Revisionen, auf die das Tag `renew4-1` und der Branch `main` zeigen: `https://git.informatik.uni-hamburg.de/tgi/Renew/-/compare/renew4-1...main`

Branch, für modulare 4.x-Versionen auf dem `main`-Branch.

Die Veränderungen auf den Hauptbranches werden im Allgemeinen durch die Qualitätssicherung durch zweistufigen Reviewprozess als bereit für einen Release angesehen.

Relevant für das Changelog sind hierbei i.d.R. nur Commits, die Plugins betreffen, welche auch tatsächlich veröffentlicht werden (s. Anhang B).

Wenn die Veränderungen zu einem Changelog zusammengefasst wurden, wird mit PO oder CPO (s. 2.4.1 Rollen) oder beiden besprochen, ob es aus ihrer Perspektive alles Relevante enthält.

Es kann bspw. sein, dass ein oder mehrere Distribution-Packages um neue Plugins erweitert werden sollen, die in vorherigen Veröffentlichungen noch nicht Teil dieser waren. In diesem Fall müssen die neuen Plugins entweder in der `plugin-names.gradle`-Datei auf dem `main`-Branch im entsprechenden Stringarray (s. 4.4.1 Gradle-Release-Tasks für 4.x) oder in der `build-dist.xml`-Datei auf dem `master`-Branch (s. Ant-Release-Targets für 2.x) ergänzt werden.

Außerdem kann es passieren, dass die Person im Releaseteam, die die Erstellung des Changelogs vorgenommen hat, die Relevanz von Änderungen nur schlecht einschätzen kann oder etwas Wichtiges übersehen hat.

Aus diesen Gründen ist es wichtig, dass an dieser Stelle Personen mit mehr Erfahrung in der RENEW-Entwicklung und einem umfangreichen Überblick über das gesamte Projekt das Changelog abnehmen.

Basierend auf dem Changelog kann auch die neue Versionsnummer entschieden werden, wenn sie nicht schon von dem CCPO vor dem Beginn der Arbeit des Releaseteams festgelegt wurde. In der Regel sollte sich die Versionsnummer recht eindeutig festlegen lassen, da sich die RENEW-Entwickler hier an Semantic Versioning (s. Abschnitt 4.2.1) halten sollten. Die Entscheidung sollte von dem Releaseteam an den CCPO kommuniziert werden, sodass dieser mögliche Einwände äußern oder zustimmen kann.

5.2 Überprüfung und Erhöhung von Versionsnummern

Vor der Bereitstellung einer neuen RENEW-Version müssen an bestimmten Stellen in der Codebasis Versionsnummern überprüft bzw. erhöht werden. Welche dies sind, wird in diesem Abschnitt beschrieben.

Weitere Versionsnummern müssen in RENEWs Dokumentation aktualisiert werden. Die diesbezügliche Vorgehensweise wird im Abschnitt „Aktualisierung der Dokumentation“ dargelegt.

Die Java-Versionsnummer sollte schon während der Entwicklung erhöht worden sein, wenn sie überhaupt erhöht werden soll, weswegen die dafür nötigen Schritte nicht beschrieben werden.

5.2.1 RENEW-Versionsnummer

Die Versionsnummer einer neuen RENEW-Version muss vor dem Bauen der zu veröffentlichenden Artefakte zunächst an zwei Stellen geändert werden.

Für 4.x-Versionen auf dem `main`-Branch müssen in der `release-tasks.gradle`-Datei die Variablen `renew_version` und `renew_version_numeric` neu gesetzt werden. Auf dem `master`-Branch muss für 2.x-Versionen das Property `renew_version` in der `build-dist.xml`-Datei neu gesetzt werden.

Für die Veröffentlichung jeder RENEW-Version muss in der `plugin.cfg`-Datei des *Gui*-Plugins der Text für den „About“-Dialog angepasst werden. Dies geschieht, indem die Ver-

sion und das Veröffentlichungsdatum im Property `de.renew.help.gui.version` und das Veröffentlichungsjahr im Property `de.renew.help.gui.content` aktualisiert werden.

5.2.2 Plugin-Versionsnummern

Die RENEW-Plugins haben jeweils eine eigene Versionsnummer. Diese werden für die einzelnen Plugins in ihrer `plugin.cfg`-Datei angegeben, die innerhalb des Ordners des jeweiligen Plugins im `src/resources`-Ordner zu finden ist.

Die Versionsnummer eines Plugins wird erhöht, wenn es zu den veröffentlichten Plugins gehört und zwischen dem vorhergehenden und dem aktuellen Release Änderungen an diesem Plugin vorgenommen wurden. Neue Plugins, die mit dem aktuellen Release zum ersten Mal veröffentlicht werden, sollten die Versionsnummer 1.0.0 erhalten.

Zusätzlich zur neuen Versionsnummer wird das Veröffentlichungsdatum `versionDate` auf das Datum gesetzt, zu dem die Veröffentlichung stattfinden soll.

Bei der Entwicklung des klassischen RENEW auf dem `master`-Branch gibt es keine klar ersichtliche Konvention für die Wahl bzw. die Erhöhung der Versionsnummern.

Auf dem `main`-Branch gilt: Auch diese Versionsnummern sollten seit dem Release von RENEW 4.0 gemäß der Semantic-Versioning-Definition (s. Abschnitt 4.2.1) gewählt werden. Aus diesem Grund wurden vor der Veröffentlichung von RENEW 4.0 alle Versionsnummern der veröffentlichten Plugins auf 2.0.0[3] festgelegt.

5.3 Aktualisierung der Dokumentation

Zu jedem Release von RENEW gehört ein Handbuch in Form einer PDF-Datei. Das Handbuch wird aus TEX-Dateien generiert.

Des Weiteren existieren zwei Readme-Dateien[4]: `README` und `README-macosx.txt`. Die `README` ist zusammen mit den Lizenzbedingungen und der Lizenz in den Distribution-Packages enthalten. Die `README-macosx.txt` ist im MacOS X Application Bundle enthalten. Die Lizenzbedingungen und die Lizenz sind in den Dateien `LICENSE` und `COPYING` aufgeführt.

Das Handbuch ist wesentlich umfangreicher als die Readme-Dateien. Abgesehen von den Lizenzdateien müssen alle hier erwähnten Dateien vor einer Veröffentlichung aktualisiert werden.

Die Dokumentationsdateien liegen im GitLab-Projekt `Renew` auf beiden Hauptbranches jeweils im Ordner `Core`. Die Abbildung 5.2 zeigt einen Verzeichnisbaum mit den für das Handbuch, die Readme-Dateien und die Lizenz-Dateien relevanten Dateien.

Das Handbuch wurde vor der Veröffentlichung von RENEW 4.0 gründlich durchgelesen und alle darin beschriebenen Funktionalitäten wurden getestet. Aufgrund des Umfangs hat dies fast die gesamte Projektzeit einer Person des Releaseteams eingenommen. Dies war jedoch für dieses Release besonders wichtig, da die modularen 4.x Versionen andere bzw. weniger Funktionalitäten beinhalten als die vorher veröffentlichten, klassischen 2.x-Versionen und sichergestellt werden musste, dass das Handbuch für die neue MAJOR-Version komplett aktualisiert wurde. Im Idealfall sollte vor jeder Veröffentlichung das gesamte Handbuch durchgelesen und auf Richtigkeit überprüft werden.

[3] Diese Versionsnummer wurde gewählt, da keins der Plugins in vorherigen Versionen eine führende Zwei hatte. Somit konnten die Versionsnummern für dieses Release vereinheitlicht werden.

[4] Eine dritte Readme-Datei namens `README.md` befindet sich im RENEW-GitLab-Projekt, diese wird zwar nicht veröffentlicht, sollte aber trotzdem zusammen mit den anderen Dateien aktualisiert werden.

```
/Renew/Core
├── doc
│   ├── images ..... Bilder für das Handbuch
│   ├── renew.tex ..... TEX-Hauptdatei des Handbuchs
│   ├── title.tex
│   ├── intro.tex
│   ├── thanks.tex
│   ├── install.tex
│   ├── history.tex
│   ├── formalism.tex
│   ├── drawing.tex
│   ├── renew.bib ..... BibTEX-Datei für Literatur
│   ├── contact.tex
│   ├── filetypes.tex
│   ├── shortcuts.tex
│   ├── license.tex
│   ├── renewdef.sty ..... TEX-Datei für Makros
│   ├── plugindefs.sty
│   ├── COPYING ..... GNU LGPL Version 2.1
│   ├── LICENSE ..... Lizenzbedingungen
│   └── README ..... README für RENEW
└── mac
    └── README-masosx.txt . RENEWs README für das MacOS X Application Bundle
```

Abbildung 5.2: Dateien für Handbuch, Readme-Dateien, und Lizenzbedingungen

Hier werden die Anpassungen und Änderungen aufgezählt, die auf jeden Fall vor einer Veröffentlichung vorgenommen werden müssen. Wenn neue Versionen beider Produktlinien zusammen veröffentlicht werden, müssen diese Schritte für beide Versionen auf beiden Hauptbranches ausgeführt werden.

Wie bereits erwähnt, müssen auch im Handbuch Versionsnummern aktualisiert werden. Die Tabelle 5.1 gibt einen Überblick darüber, welche Versionsnummern in welchen Dateien geändert werden müssen.

Datei	Zu Bearbeitendes
`Core/doc/renewdef.sty`	renewversion-[5], renewyear-[6], newinversion[7]-Makro
`Core/doc/plugindefs.sty`	Plugin-Versionsnummern in JAR-Dateinamen[8]
`Core/doc/install.tex`	Benötigte Java-Version
`Core/doc/README`	Alle Erwähnungen der aktuellen Version
`Core/mac/README-macosx.txt`	Aktuelle Version im ersten Satz

Tabelle 5.1: Zu ändernde Versionsnummern etc. in Dokumentationsdateien

[5] Bspw. für 4.1: `\newcommand{\renewversion}{4.1}`

[6] Bspw. für 4.1: `\newcommand{\renewyear}{2023}`

[7] Bspw. für 4.0: `\newcommand\newfourdotzero{\newinversion{4.0}}`

[8] In der Form `<plugin-name>-MAJOR.MINOR.PATCH.jar`, bspw. für das Makro `\PluginGui` ist es `\newcommand{\PluginGui}{de.renew.gui-2.0.0.jar}` Hierbei ist darauf zu achten, dass die hier angegebenen JAR-Dateinamen mit denen der tatsächlichen JARs in den Distribution-Packages übereinstimmen.

Da die RENEW-Versionsnummer, -Veröffentlichungsjahr und Plugin-JAR-Dateinamen (inklusive Versionsnummern) in den TEX-Dateien stets mit Makros referenziert werden, können diese in den TEX-Style-Definitionsdateien `renewdef.sty` und `plugindefs.sty` zentral abgeändert werden. Die `plugindefs.sty` wird in der `renewdef.sty` eingebunden, welche in der Hauptdatei des Handbuchs `renew.tex` eingebunden wird.

Es existiert ein Shellskript im `Core/doc`-Ordner, mit welchem die `plugindefs.sty` automatisch aktualisiert werden kann, nämlich `prepare-plugin-versions.sh`. In diesem Skript können die RENEW-Versionsnummer eingetragen und die relevanten Plugins in der Variable `list` angegeben werden, sodass die Makros für die JAR-Dateinahmen für diese Plugins neu generiert werden.

Das Skript ist nur für den `master`-Branch aktuell. Für den `main`-Branch müsste das Skript zuerst angepasst werden, bevor es genutzt werden kann. Bei der Anpassung müsste die Liste mit den veröffentlichten bzw. den im Handbuch referenzierten Plugins abgeglichen und die Struktur der JAR-Dateinamen verändert werden, da diese für 4.x-Plugins anders ist als für 2.x-Plugins.

In der `install.tex` muss die für das bevorstehende Release benötigte Java-Version im Abschnitt „Prerequisites“ eingetragen werden.

In den Readme-Dateien müssen alle Erwähnungen der aktuellen RENEW-Version und der benötigten Java-Version händisch aktualisiert werden.

Alle TEX-Dateien, die für die Bearbeitung des Handbuchs relevant sind, sind in Abbildung 5.3 dargestellt. Die rechte Seite gibt an, wofür eine Datei verwendet wird bzw. mit welcher Überschrift sie in der PDF-Datei korrespondiert. Übergeordnete Dateien binden die Dateien unter sich jeweils ein.

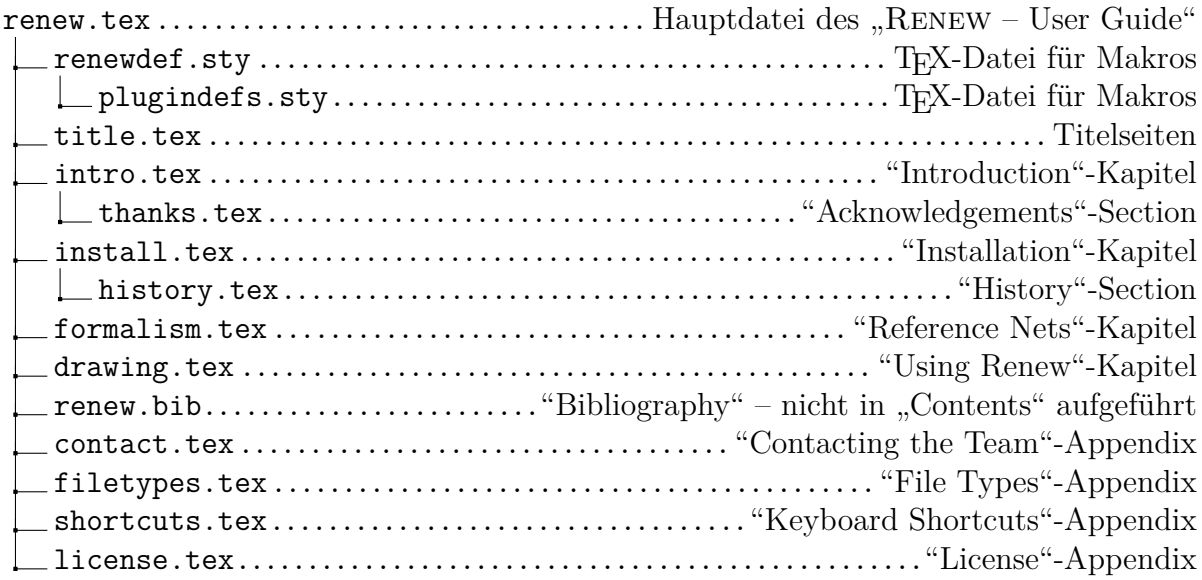


Abbildung 5.3: TEX-Dateien des Handbuchs

Personen, die an der RENEW-Entwicklung beteiligt waren bzw. sind, haben die Möglichkeit, im Handbuch als Mitwirkende aufgeführt zu werden. Dafür wird vor einer Veröffentlichung im für die Kommunikation zwischen den RENEW-Entwickelnden genutzten Mattermost[9] gefragt, wer in der Liste der Mitwirkenden aufgeführt werden möchte, wobei dies natürlich freiwillig ist.

[9] https://mattermost.com/

Die Liste muss sowohl in der `thanks.tex`-Datei, als auch in der `README`-Datei dementsprechend ergänzt werden. Die hier erwähnten Personen sollten dieselben sein, wie die in der `contributors.html`-Datei (s. Abschnitt 5.4.3).

Das Handbuch enthält einen Abschnitt mit dem Titel „Upgrade Notes“. Sie soll Nutzenden von RENEW dabei helfen, von einer älteren Version auf eine neuere zu wechseln.

In der Regel ist RENEW rückwärts-, aber nicht zwingend vorwärtskompatibel (Kummer u. a. 2023b).

Falls es für die Nutzung der zu veröffentlichenden Version relevante Hinweise bezüglich des Upgrades von einer älteren Version zu dieser gibt, sollten diese hier festgehalten werden. Diese Änderungen sind in der Datei `install.tex` vorzunehmen.

Im Abschnitt „History“ des Handbuchs ist die Release History von RENEW zusammengefasst.

In der Datei `history.tex` wird das für die zu veröffentlichende Version zusammengestellte Changelog (s. Abschnitt 5.1) hinzugefügt. Das Changelog sollte mit dem in der `history.html`-Datei für die Webseite übereinstimmen (s. Abschnitt 5.4.3).

Zusätzlich ist zu überprüfen, ob die Neuerungen, die die neue Version mit sich bringt, im Handbuch dokumentiert wurden. Sollte dies nicht der Fall sein, muss es nun nachgeholt werden.

Unter Umständen ist es dafür nötig, die an der Entwicklung von neuen Features beteiligten Personen zu kontaktieren, sollten die Neuerungen nicht gut genug im Quelltext oder der Readme-Datei des betroffenen Plugins dokumentiert sein.

Die Dokumentationen neuer Features werden mit dem Makro `\newXdotY{…}`[10] umgeben, wodurch für erfahrene RENEW-Nutzende die Dokumentation von Neuerungen schneller zu erkennen ist. In Abbildung 5.4 ist die durch das entsprechende Makro erzeugte Markierung zu sehen, mit der Neuerungen in RENEW 4.1 hervorgehoben wurden.

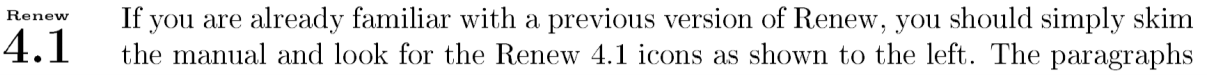


Abbildung 5.4: Kennzeichnung im Handbuch für Neuerungen in RENEW 4.1

Dieses Makro muss außerdem in der Datei `intro.tex` anstelle des Makros der vorherigen Version ersetzt werden. Alle derartigen Makros von vorherigen Versionen müssen entfernt werden.

Im Handbuch sind auch Bugs dokumentiert. Durch die Nutzung des Makros`\bug{…}` werden diese gekennzeichnet.

Neue Bugs müssen an der entsprechenden Stelle dokumentiert werden, außerdem sollten die Beschreibungen behobener Bugs aus dem Handbuch entfernt werden.

RENEWs Lizenzbedingungen werden im Anhang des Handbuchs wiedergegeben und sind in der Datei `license.tex` festgehalten.

Aus dieser Datei wird auch die `LICENSE`-Datei von Gradle generiert.

Die Lizenzbedingungen im Handbuch sollten mit jenen in der `license.html` für die Webseite festgehaltenen übereinstimmen (s. Abschnitt 5.4.3).

[10] X steht für die MAJOR-Versionsnummer, Y für die MINOR-Versionsnummer; jeweils als englisches Wort.

In der `COPYING`-Datei ist eine Kopie der Lizenz *GNU LGPL Version 2.1*, unter welcher RENEW veröffentlicht wird, zu finden. Sie müsste nur aktualisiert werden, falls RENEW unter einer anderen Lizenz veröffentlicht werden soll.

Wenn alle Änderungen am Handbuch und in den Readme-Dateien vorgenommen wurden, sollten die drei Dateien Korrektur gelesen werden. Zu diesem Zweck kann das Handbuch als PDF-Datei aus den TEX-Dateien mit dem Befehl `./gradlew clean buildGuides` generiert werden.

Im Anschluss können die Änderungen am Handbuch nach den üblichen Reviews in den Hauptbranch übernommen werden.

5.4 Vorbereitung der Webseiten

Für die RENEW-Webseite existiert ein eigenes GitLab-Projekt namens `renew-web`[11], dessen Verzeichnisbaum in Abbildung 5.5 dargestellt ist[12].

Um eine neue Version zu veröffentlichen müssen in `renew-web` einige Vorbereitungen getroffen werden.

In den folgenden Abschnitten werden die Schritte „Erstellung eines Prepare-Branchs“, „Generierung eines Versionsordners“ und „Händische Anpassung von HTML-Dateien“ genauer erläutert.

5.4.1 Erstellung eines Prepare-Branchs

Wenn eine neue Version veröffentlicht werden soll, wird zunächst ein neuer Branch vom Hauptbranch `master` von `renew-web` abgezweigt.

Dieser erhält einen Namen der Form `prepare-[VERSION]`, wobei `[VERSION]` durch die entsprechende Versionsnummer ersetzt wird[13]. Werden eine neue klassische und eine neue modulare Version von RENEW zusammen veröffentlicht, dann kann, um dies ersichtlich zu machen, dem Namen ein `-dualversion` angehängt werden.

5.4.2 Generierung eines Versionsordners

Nun wird die `packages.zip`-Datei, welche mit Hilfe der CI/CD-Pipeline des RENEW-Projekts generiert und als Artefakt gespeichert wurde, über die GitLab GUI heruntergeladen und auf dem neu erstellten Prepare-Branch im Stammverzeichnis `renew-web` extrahiert.

Für die neue RENEW-Version wird eine JSON[14]-Kontextdatei im `templates`-Ordner angelegt, in der die vier Parameter `renewversion`, `versiontype`, `javaversion` und `packages` festgelegt werden.

Diese Datei wird `renew[VERSION]context.json` genannt, wobei auch hier `[VERSION]` durch die entsprechende Versionsnummer ersetzt wird.

Der `renewversion` wird auch diese Versionsnummer als String zugewiesen. Der Variable `versiontype` wird entweder `classic`, wenn es eine 2.x-Version ist, oder `modular`, wenn

[11] `https://git.informatik.uni-hamburg.de/tgi/renew-web`
[12] Einige Dateien wie .gitignore, README.md, Bilder etc. werden der Übersicht halber nicht aufgelistet.
[13] Bspw. für 4.1: `prepare-4.1`
[14] **Java**Script **O**bject **N**otation

```
renew-web
├── 1.0
├── …
├── 4.1 ..... Versionsordner für Renew 4.1
│   ├── extraPlugins4.1 ..... Ordner mit Extra-Plugins für Renew 4.1
│   ├── ide ..... Ordner für Distribution-Package IDE von Renew 4.1
│   ├── plugins4.1 ..... Order mit optionalen Plugins für Renew 4.1
│   ├── Renew-4.1.dmg ..... MacOS X Application Bundle für Renew 4.1
│   ├── download.html ..... "Download Renew 4.1"
│   ├── install.html ..... „Install Renew 4.1"
│   ├── plugins4.1.html ..... (optionale) „Plug-ins for Renew 4.1"
│   ├── renew4.1.pdf ..... Handbuch für Renew 4.1
│   ├── renew4.1base.zip ..... ZIP mit Base-Package von Renew 4.1
│   └── renew4.1source.zip ..... ZIP mit Source-Package von Renew 4.1
├── templates
│   ├── classic ..... Mustache-Templates für klassische Versionen
│   ├── modular ..... Mustache-Templates für modulare Versionen
│   │   ├── download.mustache
│   │   ├── extraPlugins.mustache
│   │   ├── install.mustache
│   │   └── plugins.mustache
│   ├── …
│   └── renew4.1context.json ..... Kontextdatei für Renew 4.1
├── archive.html ..... „Version Archive"
├── bibliography.html ..... „Renew Bibliography"
├── contributors.html ..... „Contributors"
├── createfromtemplates.py ..... Skript für die Generierung eines Versionsordners
├── faq.html ..... „Frequently Asked Questions"
├── highlights.html ..... „Highlights"
├── history.html ..... „Release History"
├── issues.html ..... „Known Issues"
├── license.html ..... „License Terms"
├── old-issues.html ..... „Known Issues in old Versions"
├── renew.html ..... Startseite von http://www.renew.de/
└── update-dates-with-sed.sh Skript zur Aktualisierung der „Last modified"-Einträge
```

Abbildung 5.5: Verzeichnisbaum von `renew-web`

es eine 4.x-Version ist, als String zugewiesen. Der `javaversion` wird die benötigte Java-LTS-Version als String zugewiesen; bei RENEW 2.6, 4.0 und 4.1 ist dies 11 gewesen, bei zukünftigen modularen Versionen wäre es 17. Den `packages` wird ein String-Array zugewiesen, welches die verschiedenen Distribution-Packages spezifiziert; für 4.0 und 4.1 ist dieses lediglich `["ide"]`, für 2.6 `["analysis","pd","ide","ccpn"]`.

Mit dem Pythonskript `createfromtemplates.py` kann nun ein Versionsordner aus dem extrahierten `packages`-Ordner und der neuen JSON-Datei generiert werden. Wichtig ist hierbei, dass kein Versionsordner für die neue Version bereits vorhanden ist.

Für die Ausführung des Skripts wird Python 3.8+[15] und außerdem Pystache[16], da für die

[15] https://www.python.org/

[16] Mustache-Implementation von Python: https://pypi.org/project/pystache/

Generierung von HTML[17]-Dateien Mustache[18]-Templates verwendet werden, benötigt. Des Weiteren ist das Skript für die Ausführung auf Linuxsystemen ausgelegt.

Das Skript wird mittels folgenden Befehls im `renew-web`-Ordner aufgerufen, wobei auch hier `[VERSION]` durch die jeweilige Versionsnummer ersetzt wird:

```
./createfromtemplates.py templates/renew[VERSION]context.json
```

Für die Generierung der Ordner für die zusätzlichen Distribution-Packages ist dabei wichtig, dass eine `package.json`-Datei für das jeweilige Distribution-Package existiert und diese korrekt von der CI/CD-Pipeline generiert wurde.

Die entsprechende Datei kann für das Distribution-Package Renew IDE bspw. im extrahierten `packages`-Ordner im `ide`-Unterordner gefunden und überprüft werden.

Sollte eine `package.json`-Datei falsch sein, können die korrespondierenden Dateien im RENEW-GitLab-Projekt korrigiert werden. Für 2.x-Versionen liegen entsprechende XML[19]-Dateien im `master`-Branch im `productline`-Ordner; für 4.x-Versionen sind die entsprechenden Gradle-Tasks im `modular-master`- bzw. inzwischen im `main`-Branch im Stammverzeichnis in der `release-tasks.gradle`-Datei zu finden.

Nach erfolgreicher Generierung sollte es einen Versionsordner für die neue Version geben. In diesem wurden von dem Skript folgendes angelegt:

- ein ZIP-Ordner mit der Base-Distribution,
- ein ZIP-Ordner mit dem Quelltext von RENEW,
- eine DMG[20]-Datei für die Installation unter MacOS mit allen optionalen Plugins,
- das Handbuch als PDF-Datei,
- ein `plugins`-Ordner, der optionale Plugins für das Release enthält,
- ein `extraPlugins`-Ordner, welcher noch mehr optionale Plugins enthält, die allerdings nicht offiziell veröffentlicht werden[21],
- Unterordner für die Distribution-Packages und
- HTML-Dateien für die Seiten „Download Renew [VERSION]“, „Install Renew [VERSION]“, „Plug-ins for Release [VERSION]“ und „Extra Plug-ins for Release [VERSION]“[22]

5.4.3 Händische Anpassung von HTML-Dateien

Zusätzlich zu der Generierung der versionsspezifischen HTML-Dateien müssen weitere Dateien händisch angepasst werden.

Das Skript, welches zur Generierung des Versionsordners verwendet wird, gibt am Ende zusätzlich auf der Kommandozeile die Erinnerung daran aus, dass die HTML-Dateien, welche im `renew-web`-Stammverzeichnis liegen, händisch aktualisiert werden müssen. Folgende Dateien müssen immer angepasst werden: `renew.html`, `history.html`, `archive.html`,

[17] **Hyper**text **M**arkup **L**anguage
[18] https://github.com/mustache/mustache/
[19] E**x**tensible **M**arkup **L**anguage
[20] Apple **D**isk I**mag**e
[21] Diese sind für 4.1 unter `https://www2.informatik.uni-hamburg.de/TGI/renew/4.1/extraPlugins4.1/` und für andere Versionen unter analogen URLs trotzdem öffentlich einsehbar. Die URLs werden aber nicht auf der Webseite explizit verlinkt.
[22] Die HTML-Datei für die extra Plugins liegt im `extraPlugins`-Ordner und wird, wie bereits erwähnt, nirgends auf `http://www.renew.de/` verlinkt.

`issues.html`, `old-issues.html` und `license.html`[23]. Die nötigen Anpassungen in den einzelnen Dateien werden im Folgenden beschrieben.

In der Datei für die Startseite, `renew.html`, müssen alle Erwähnungen im Text und alle `href`-Referenzen auf die aktuelle Version bzw. aktuellen Versionen (klassisch und modular) gesetzt und die Veröffentlichungsdaten aktualisiert werden. In Abbildung 5.6 ist die Startseite zu sehen, wie sie nach der Veröffentlichung von RENEW 4.1 aussieht. Für die beiden aktuellen Versionen von RENEW sind hier jeweils die Downloads, Handbücher und Installationsanweisungen verlinkt. Zusätzlich sind Verlinkungen auf alle bereitgestellten, weiterführenden Informationen bzgl. RENEW zu finden.

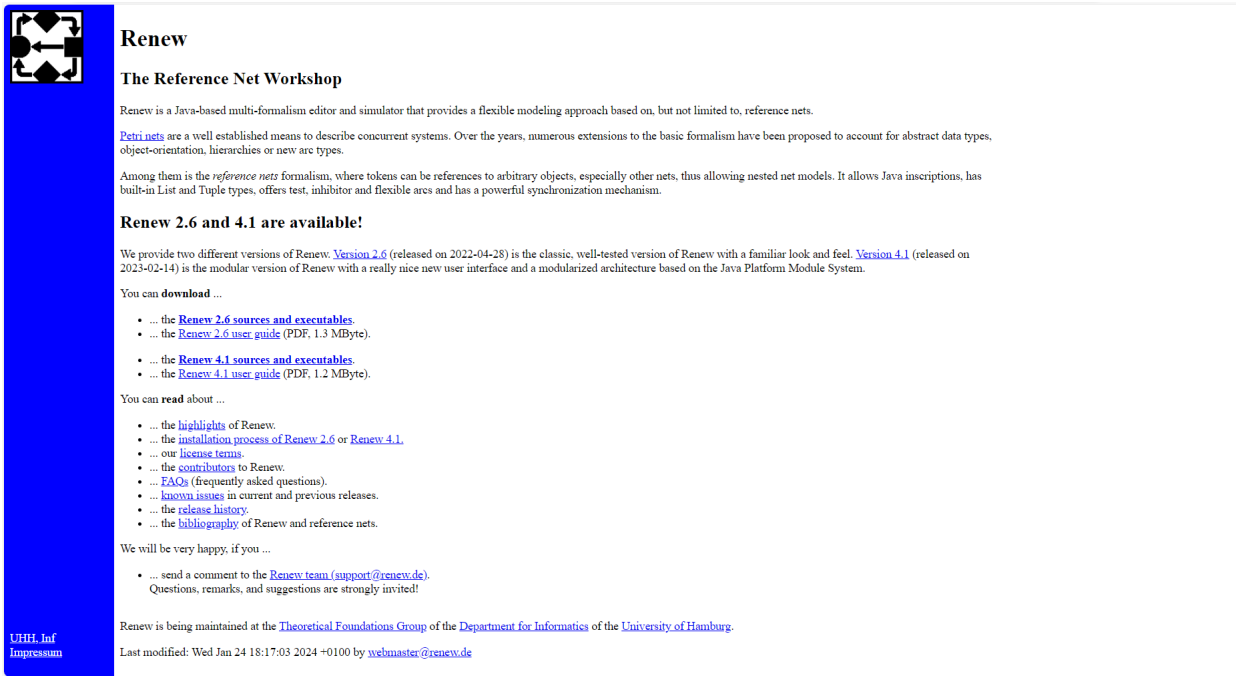


Abbildung 5.6: Startseite von `http://www.renew.de/` mit der jeweils aktuellen Version der klassischen und der modularen Variante

In der `history.html` wird die „Release History“ aufgelistet. Hier muss die neue Version ganz oben mit der Versionsnummer als `id` eingefügt werden und darunter das Veröffentlichungsdatum und kurz zusammengefasste Informationen zu Veränderungen, Neuerungen und Relevantem für Entwickelnde eingetragen werden.
Dieses Changelog sollte mit dem im Handbuch übereinstimmen (s. Abschnitt 5.3).

In der `archive.html` werden alle Versionen von RENEW mit Verlinkungen auf ihre jeweilige Downloadseite aufgelistet. In der Liste muss ganz oben die neue Version mit einer `href`-Referenz auf ihre `download.html` eingefügt werden[24].

In den Dateien `issues.html` und `old-issues.html` werden „Known Issues“ der aktuellen bzw. der alten Versionen von RENEW dokumentiert.

[23] An die händische Aktualisierung der `license.html` erinnert das Skript nicht, jedoch ist dies nötig.

[24] Das Format der `archive.html` wurde nach der Veröffentlichung von RENEW 4.1 angepasst. Zuvor wurden auf dieser Seite von den alten Versionen jeweils direkt das Base-Package im ZIP-Format, die optionalen Plug-ins, die DMG-Datei und das Source-Package im ZIP-Format verlinkt. Die Anpassung erleichtert zum einen die Wartung und zum anderen sind nun auch die weiteren Distribution-Packages dieser Versionen weiterhin für Nutzende auffindbar.

Die „Known Issues“ bestehen aus der Beschreibung von bekannten Problemen und Bugs, die die Software enthält, einer Angabe der Umgebung, in der die Probleme auftreten, und der Ursache, sofern diese bekannt ist. Außerdem wird eine Übergangslösung für das Problem angegeben, wenn eine solche bereits entdeckt wurde.

Bei der Veröffentlichung einer neuen Version müssen die bekannten Probleme der vorhergehenden Version, die durch die neue Version abgelöst wird, aus der `issues.html` in die `old-issues.html` übertragen werden. Die Probleme der neuen Version werden in die `issues.html` eingetragen.

In der `license.html`-Datei werden die Lizenzbedingungen für die Nutzung von RENEW beschrieben.

Im Abschnitt „Contributed Parts“ werden Softwareteile aufgelistet, welche von externen Personen entwickelt wurden und von RENEW genutzt werden. Sollte neue externe Software von der neuen RENEW-Version verwendet werden, die von vorherigen Releases nicht genutzt wurde, müssen die Lizenzbedingungen für diese Software an dieser Stelle hinzugefügt werden. Bei der Veröffentlichung von RENEW 4.0 mussten hier die Verweise auf die Lizenzbedingungen von DockingFrames[25] und vom Gradle-Wrapper (s. Abschnitt 4.4.1) ergänzt werden.

Im Abschnitt „Original Parts“ werden die Lizenzbedingungen für die Teile von RENEW festgelegt, die nicht auf der Arbeit anderer aufbaut. Hier muss bei jeder neuen Veröffentlichung das Copyright-Datum aktualisiert werden.

Die hier dokumentierten Lizenzbedingungen sollten mit denen im Handbuch übereinstimmen (s. Abschnitt 5.3).

Die anderen HTML-Dateien, nämlich `bibliography.html`, `contributors.html`, `highlights.html` und `faq.html`, müssen selten angefasst werden, sollten aber dennoch überprüft werden.

In der Datei `bibliography.html` werden die wichtigsten wissenschaftlichen Publikationen zu den Themen Referenznetzformalismus und RENEW aufgelistet.

Diese Seite wurde während der von dieser Arbeit begleiteten Veröffentlichungen nicht inhaltlich verändert. Etwaige Änderungen müssten mit dem CCPO abgesprochen werden.

In der `contributors.html`-Datei sind Personen aufgelistet, die sich an der Entwicklung von RENEW beteiligt haben. Da an neuen Versionen i.d.R. auch Personen mitgewirkt haben, die hier noch nicht aufgeführt werden, können diese hier hinzugefügt werden, sollte es von den entsprechenden Personen gewünscht sein.

Die Liste sollte mit der im Handbuch übereinstimmen (s. Abschnitt 5.3).

Vorteile von und Anwendungsmöglichkeiten für RENEW werden in der `highlights.html`-Datei beschrieben.

Diese Seite wurde während der von dieser Arbeit begleiteten Veröffentlichungen nicht inhaltlich verändert. Etwaige Änderungen müssten mit dem CCPO abgesprochen werden.

Häufig zu RENEW gestellte Fragen, also „Frequently Asked Questions“, werden in der `faq.html`-Datei aufgelistet und beantwortet.

Diese Seite wurde während der von dieser Arbeit begleiteten Veröffentlichungen nicht inhaltlich verändert. Etwaige Änderungen müssten mit dem CCPO abgesprochen werden.

[25] http://www.docking-frames.org/

Mit jeder händischen Anpassung einer HTML-Datei müssen am Ende das Datum und die Uhrzeit im *Last modified*-Eintrag der Datei aktualisiert werden.

Während der von dieser Arbeit begleiteten Veröffentlichungen wurde diese Aktualisierung händisch vorgenommen.

Seitdem wurde das Shellskript `update-dates-with-sed.sh` von einem Hauptentwickler geschrieben, welches für die Aktualisierung verwendet werden kann, nachdem Änderungen an den HTML-Dateien committet wurden. Für die Ausführung wird *GNU sed*[26] benötigt.

5.5 Zweistufige Qualitätssicherung

Nachdem nun die generierten und die händisch angepassten Dateien auf dem Prepare-Branch liegen und von den entwickelnden Personen, die das Release vorbereiten, als fertig angesehen werden, sollten die veränderten HTML-Seiten überprüft, die Dokumentationsdateien begutachtet und die Distribution-Packages und Plugins der neuen Version auf verschiedenen Betriebssystemen – Linux, Windows und MacOS – getestet werden.

Dies geschieht zunächst durch das Releaseteam selbst. Sollten nicht alle Betriebssysteme von diesem Personenkreis abgedeckt werden können, können andere Projektmitglieder für das Testen angefragt werden.

Da für diese Überprüfung keine Konventionen bestanden, wurde innerhalb dieser Arbeit eine Release Review Checkliste entworfen, welche in Anhang D zu finden ist. Der Nutzen und die Entstehung der Checkliste wird in Abschnitt 6.2.1 beschrieben. Sie sollte von zukünftigen Releaseteams verwendet und erweitert werden.

Wenn die Releasevorbereitungen auf dem Prepare-Branch hinreichend getestet wurden, wird ein*e Hauptentwickler*in[27] benachrichtigt.

Ein*e Hauptentwickler*in hat die nötige Erfahrung in der RENEW-Entwicklung und umfassende Kenntnisse von der Software, um zu beurteilen, ob das Release bereit für die Veröffentlichung auf `http://www.renew.de/` ist.

Wenn die Beurteilung in diesem Review positiv ausfällt, kann die Version veröffentlicht werden.

Sollte sie negativ ausfallen, werden Anmerkungen von dem*der Hauptentwickler*in gemacht, was noch problematisch oder fehlerhaft ist. Je nach Art des Problems muss zu unterschiedlichen Schritten im Veröffentlichungsprozess zurückgesprungen und es dort behoben werden.

Wenn Fehler in HTML-Seiten bemerkt wurden, können diese direkt im Projekt `renew-web` überarbeitet werden. Sollte die Überarbeitung generierte HTML-Dateien im Versionsordner betreffen, so sollten die zugehörigen Mustache-Templates auch überarbeitet werden.

Wenn allerdings Fehler im Handbuch, in den Distribution-Packages von RENEW oder innerhalb anderer generierter Artefakte gefunden wurden, müssen diese im `Renew`-GitLab-Projekt behoben werden. Dies hat zur Folge, dass auch die `packages.zip`-Datei mit Hilfe der CI/CD-Pipeline neu generiert werden muss und danach in `renew-web` der Versionsordner gelöscht und mit dem Pythonskript aus den entzippten `packages` neu generiert werden muss.

Bei der Überarbeitung von Fehlern im Handbuch sollte immer überprüft werden, ob der Teil des Handbuchs eine Entsprechung auf der Webseite hat, damit diese ebenfalls überarbeitet wird. Das gilt andersherum ebenso. In der Regel ist dies bei Änderungen in

[26] "sed" steht für **s**tream **ed**itor; `https://www.gnu.org/software/sed/`

[27] Für die Releases von RENEW 2.6, 4.0 und 4.1 war dies Michael Haustermann.

den folgenden Dateien zu beachten: `history.html` bzw. `history.tex`, `license.html` bzw. `license.tex`, `contributors.html` bzw. `thanks.tex`.

5.6 Deployment des Release

Ein neues Release wird auf die öffentliche Webseite `http://www.renew.de/` ausgeliefert. Wie dies erreicht wird, ist im Abschnitt „Deployment auf TGI-Server“ beschrieben.

Seit der Veröffentlichung von RENEW 4.1 wird zusätzlich ein experimentelles Flatpak-Release der Software bereitgestellt. Erläuterungen zu diesem Release sind in Abschnitt „Deployment auf Flathub“ zu finden.

5.6.1 Deployment auf TGI-Server

Bevor die Änderungen von `renew-web` auf die öffentliche Webseite `http://www.renew.de/` ausgeliefert (engl.: deployed) werden, wird der Prepare-Branch in den `master`-Branch gemergt.

Dies passiert mit „squash and merge“, welches eine Mergestrategie ist, durch die alle Commits, die auf dem zu mergenden Prepare-Branch ausgeübt wurden, zu einem einzigen Commit zusammengefasst werden (GitLab Inc. 2024o). Der Prepare-Branch kann hiernach gelöscht werden.

Auf dem `master`-Branch sollte das Release nach erfolgreichem Deployment auf die öffentliche Webseite mit einem Tag[28] versehen werden.

Für das Deployment auf `http://www.renew.de/` wurden auf Anfrage die folgenden Informationen von Michael Haustermann in der `README.md`[29] von `renew-web` ergänzt:

Auf der Maschine `www2.informatik.uni-hamburg.de` befindet sich ein Klon des renew-web-Projekts im Verzeichnis `/export/home/wwwtgi/WWW/renew`. Dieser Klon ist vom Nutzer „wwwtgi“ ausgecheckt. Der Inhalt dieses lokalen Klons ist öffentlich zugänglich über den Webserver (`https://www2.informatik.uni-hamburg.de/TGI/renew/` bzw. `https://www.informatik.uni-hamburg.de/TGI/renew/`).

`http://renew.de` ist eine Weiterleitung auf diese URL [30]. Diese Weiterleitung ist insofern dynamisch, als dass z.B. `http://renew.de/4.0/download.html` auf `https://www2.informatik.uni-hamburg.de/TGI/renew/4.0/download.html` weiterleitet.

Für `http://www.renew.de/` scheint https nicht richtig konfiguriert zu sein.

Es ist wichtig zu wissen, dass alles, was auf `www2.informatik.uni-hamburg.de` ausgecheckt wird, potenziell über das Internet zugänglich ist. In dem Repository sollten also keine Geheimnisse hinterlegt werden.

Der Klon auf `www2.informatik.uni-hamburg.de` ist als „sparse-checkout“[31] konfiguriert. Durch diese Konfiguration sind bestimmte Dateien, wie das Pythonskript und die Mustache-Templates, in diesem Klon nicht vorhanden.

Für das Git-Repository (also den `.git`-Ordner) hat nur der Besitzer „wwwtgi“ Zugriffsrechte. Der Webserver läuft als ein anderer Nutzer, wodurch vermieden wird, dass auf das Git-Repository selbst über den Webserver zugegriffen wird.

[28] Tags in `renew-web`: `https://git.informatik.uni-hamburg.de/tgi/renew-web/-/tags`
[29] Dort sind sie in Englisch verfasst, für die Wiedergabe in dieser Arbeit wurden sie übersetzt.
[30] Michael Haustermann vermutet, dass nur Olaf Kummer Zugang zu der Konfiguration hat.
[31] Dokumentation von git-sparse-checkout: `https://git-scm.com/docs/git-sparse-checkout`

Um die Webseite zu veröffentlichen muss lediglich die gewünschte Revision auf der Maschine `www2.informatik.uni-hamburg.de` gefetcht und ausgecheckt werden.

Der Zugriff auf den Server erfolgt also per ssh mit dem Nutzer „wwwtgi“. Das Passwort für diesen Nutzer wird nicht großzügig verteilt, allerdings ist es möglich Projektbeteiligten Zugriff per öffentlichem ssh-Schlüssel zu geben. Die Verantwortung für die Verteilung von Zugriffsrechten obliegt dem CCPO.

Die Konfiguration des Webservers wurde in den von dieser Arbeit begleiteten Veröffentlichungen nicht verändert.

Die nötigen Schritte für das Deployment wurden nicht von Studierenden sondern von Michael Haustermann ausgeführt.

5.6.2 Deployment auf Flathub

Für RENEW 4.1 wurde erstmals ein Flatpak[32]-Release der Software auf Flathub[33] bereitgestellt[34]. *Flatpak* ist ein Paketformat, welches eine einfache Bereitstellung von Anwendungen für alle Desktop-Versionen von Linux erlaubt.

Das Flatpak-Release wurde bisher von Michael Haustermann erstellt und ist experimentell. Für die Aktualisierung des Flatpak-Releases müssen sowohl in `renew-web` als auch in einem Flathub-GitHub-Repository[35] Änderungen vorgenommen werden. Auf dieses Repository hat derzeit nur Michael Haustermann Zugriff. Genauere Hinweise für die Aktualisierung des Flatpak-Release sind im Abschnitt „Flatpak“ der `README.md` von `renew-web` zu finden.

[32] `https://www.flatpak.org/`
[33] `https://flathub.org/`
[34] RENEW auf Flathub: `https://flathub.org/apps/de.renew.Renew`
[35] RENEWs Flathub-GitHub-Repository: `https://github.com/flathub/de.renew.Renew`

Kapitel 6

Fragestellungen im Rahmen der Veröffentlichung

Um Releaseprozesse zuverlässiger zu gestalten und zu automatisieren, müssen diese zunächst durchdrungen und dokumentiert werden (Adams u. a. 2015; Barnett 2020). Das gilt für Releaseprozesse im Allgemeinen, also auch für diesen Prozess innerhalb der RENEW-Entwicklung. Die Dokumentation des Prozesses erfolgte in den vorhergehenden Kapiteln 4 und 5 durch die strukturierte Wiedergabe der Erfahrungen, die bei der Vorbereitung und der Veröffentlichung der Releases von RENEW 2.6, 4.0 und 4.1 gesammelt wurden.

In diesem Kapitel werden darauf aufbauend beobachtete Herausforderungen und Störungen in den Releasevorbereitungen festgehalten und Vorschläge gemacht, wie diesen begegnet werden kann, sodass zukünftige Iterationen des Releaseprozesses störungsfreier durchgeführt werden können bzw. von einem höheren Automatisierungsgrad profitieren. Diese Dokumentation der Problematiken und die Betrachtungen möglicher Anpassungen zeigen, welche Fallstricke bei der Veröffentlichung einer komplexen, im universitären Kontext entwickelten Open-Source-Software wie RENEW, deren Releaseprozess größtenteils manuell durchgeführt wird, auftreten können und wie mit ihnen umgegangen werden kann.

Die Automatisierung des gesamten Releaseprozesses ist erstrebenswert, denn wenn er nicht vollautomatisiert ist, ist der gesamte Prozess fehleranfälliger, undurchsichtig für Personen, die ihn noch nie ausgeführt haben, abhängig von wenigen Expert*innen, repetitiv, zeitaufwändig, teuer und schwer zu testen bzw. zu kontrollieren (Humble und Farley 2011, S.5ff). Humble und Farley vertreten den Standpunkt, dass so viel wie möglich innerhalb von Releaseprozessen schrittweise automatisiert werden sollte (Humble und Farley 2011, S.25). Andere Autoren merken an, dass sorgfältig abgewogen werden sollte, ob die Ansprüche an den Veröffentlichungsprozess vollautomatisiert erfüllt werden können (Dwyer 2024; Reed 2020).

Zusätzlich zu Automatisierungsvorschlägen werden im Folgenden auch Vorschläge gemacht, die nicht direkt auf eine Automatisierung abzielen, da ein vollautomatisierter Releaseprozesses ein langfristiges Ziel ist und manche Schritte nicht automatisiert werden können bzw. sollten. Hierbei geht es darum, Konventionen zu schaffen, organisatorische bzw. strukturelle Hürden nach Möglichkeit zu beseitigen und nötige Schritte vor einer möglichen Automatisierung aufzuzeigen.

Die beleuchteten Fragestellungen sind in Abschnitten zu den Themenbereichen Technische Schulden, Umsetzung, Organisation und Automatisierung gegliedert.

Im letzten Abschnitt dieses Kapitels werden Gegebenheiten und Vorgänge innerhalb des Releaseprozesses einer im universitären Kontext entwickelten Software wie RENEW erfasst, die zwar als Hürde innerhalb des Prozesses identifiziert wurden, sich aber nicht än-

dern lassen bzw. wirklich nur händisch vorgenommen werden können.

6.1 Technische Schulden

Während des Softwareentwicklungsprozesses getroffene Entscheidungen, aufgrund derer die spätere Weiterentwicklung erschwert bzw. aufwändiger wird, nennt man *technische Schulden* (Rouse 2017).

Die im Kontext von RENEWs Releaseprozess relevanten technischen Schulden der lückenhaften Testabdeckung und fehlenden Analysewerkzeuge werden im Folgenden reflektiert.

6.1.1 Testabdeckung

Problembeschreibung

Beim Testen von Software soll gezeigt werden, dass eine Software so funktioniert, wie sie funktionieren soll, und es sollen Fehler vor der Benutzung aufgedeckt werden (Sommerville 2018, Kap. 8). Tests sind Teil der Qualitätssicherungsmaßnahmen in der Softwareentwicklung.

Al Shriteh hat in seiner Arbeit die vorhandenen Tests von RENEW beschrieben, eine Teststrategie entwickelt und für einige Module und Klassen neue Tests geschrieben.

Trotz dieser Anstrengungen ist die Codebasis von RENEW immer noch nicht durch Tests abgedeckt. Des Weiteren fehlen Tests für den den Bauprozess und die Benutzeroberfläche (Al Shriteh 2022).

Aufgrund der unzureichenden automatisierten Tests konnte bei den Veröffentlichungen das Vertrauen darin, dass eine zu veröffentlichende Version den Ansprüchen an die Funktionalität und Zuverlässigkeit genügt, nur durch händisches Testen der Software durch die Entwickelnden hergestellt werden.

Dieser Umstand stellt eine Lücke in den Qualitätssicherungsmaßnahmen dar und steht zusätzlich einer Automatisierung des Veröffentlichungsprozesses im Weg. Karvonen u. a. nennen als zwei der Voraussetzungen für den Einsatz von Continuous Deployment, welches eine solche Automatisierung der Veröffentlichung wäre, das Vorhandensein von automatisierten Tests, welche einen signifikanten Teil der Kernfunktionalität der Software sicherstellen können, und das Integrieren von Akzeptanztests durch Kunden (engl.: customer acceptance testing) in die Deployment-Pipeline (Karvonen u. a. 2017). Auch Michlmayr u. a. betont die Wichtigkeit von Testfeedback durch Nutzende. Aus diesem seien die signifikantesten Erkenntnisse zu erlangen (Michlmayr u. a. 2007).

Vorschläge

Es ist für die Qualitätssicherung zukünftiger Releases also erstrebenswert, die Testabdeckung des Legacy-Codes auszuweiten, indem die von Al Shriteh erarbeitete Teststrategie auf ganz RENEW angewendet wird.

Da die Codebasis von RENEW bereits sehr groß ist und verhältnismäßig zu seiner Komplexität wenige Tests besitzt (Al Shriteh 2022), wird das Erreichen einer vollständigen Testabdeckung allerdings sehr lange dauern.

Neuer Code muss laut Konvention korrespondierende Tests haben und Merge Requests werden sowohl teamintern auf Funktionalität und Einhaltung der Konventionen geprüft

als auch von erfahreneren RENEW-Entwickelnden (s. 4.3.2 Qualitätssicherung durch zweistufigen Reviewprozess).

Außerdem sollten Alternativen in Betracht kommen, mit denen Tests schneller erstellt werden können als allein durch Entwickelnde. Eine Möglichkeit stellt die evtl. Nutzung von generativer KI[1] für die Erstellung von Tests dar (Langemann 2023; Shringi und Singh 2024).

Zur Einholung von Nutzendenfeedback wäre es u.U. ratsam, Studierende, welche an Modulen, in denen RENEW genutzt wird, teilnehmen, dazu zu verpflichten, die zuletzt veröffentlichte Version zu verwenden und Feedback zu der Software zu geben.

So könnten noch vorhandene Fehler vermutlich effektiver gefunden werden.

6.1.2 Analysewerkzeug

Problembeschreibung

Sonargraph[2] ist ein Werkzeug, welches für die statische Codeanalyse eingesetzt werden kann (Renew-Development-Team 2023c). Diverse Metriken, mit denen bspw. die Komplexität, die Verschachtelung und das Vorhandensein von Redundanzen im Quelltext gemessen werden, können mit Sonargraph erhoben und angezeigt werden (Renew-Development-Team 2023a).

Zur Zeit werden innerhalb des Releaseprozesses keine der Metriken genutzt, um die Qualität eines Release zu messen, bzw. Grenzwerte für die Metriken beachtet, die vor einer Veröffentlichung erreicht werden müssen.

Vorschlag

Innerhalb von Entwicklungs- und Releaseprozessen ist es hilfreich, gewisse Metriken zu sammeln, zu analysieren und auf Basis dieser Maßnahmen zu ergreifen (Bryant und Marín-Pérez 2018, S.26).

Innerhalb der RENEW-Entwicklung wird bereits an der Integration von Sonargraph in den Entwicklungsprozess gearbeitet. Es sollte darauffolgend ermittelt werden, inwiefern die erhobenen Metriken in die CI/CD-Pipeline integriert werden können, sodass die Ergebnisse zum automatischen Feedback auf gepushte Commits gehören und insbesondere bei der Erstellung eines Release berücksichtigt werden. Diese Qualitätssicherungsmaßnahme könnte das Vertrauen in RENEWs Codequalität erhöhen.

6.2 Umsetzung

Innerhalb der Umsetzung des Veröffentlichungsprozesses von RENEW bestehen gewisse Herausforderungen, die während der Durchführung der Releases beobachtet wurden.

Diese betreffen fehlende Review-Konventionen der Releasevorbereitungen, Fehler bei der Versionierung von Plugins, fehlerhafte Pluginbezeichnungen im Handbuch, voneinander abweichende Lizenzbedingungen, redundante und veraltete Dokumentation.

[1] **K**ünstliche **I**ntelligenz

[2] https://www.hello2morrow.com/products/sonargraph

6.2.1 Review-Konventionen der Releasevorbereitungen

Problembeschreibung

Teil des Veröffentlichungsprozesses von RENEW ist es, einen Review der Vorbereitungen – bzgl. des Release selbst, zu dem der Code, die Builds, die Readme-Dateien und das Handbuch gehören, und der aktualisierten Webseite – innerhalb des Releaseteams durchzuführen.

Dieser wichtige Schritt hat jedoch bisher keine Konventionen und ist dadurch willkürlich. Die Gründlichkeit des Reviews hängt von den beteiligten Personen und deren Überblick über das Release ab.

So wird nur getestet, was den Reviewenden relevant erscheint, bspw. dass RENEW gestartet werden kann, ein Netz gemalt und gespeichert werden kann oder dass optionale Plugins hinzugefügt und geladen werden können. Die relevanten Stellen des Handbuchs und der Readme-Dateien sind hoffentlich bekannt und werden kontrolliert. Gleiches gilt für die Dateien der Webseite.

Vorschlag

Um die nötigen Aspekte des händisch durchzuführenden Release-Reviews zu bestimmen, wurde die Qualitätssicherung innerhalb des Releaseprozesses mit einem Hauptentwickler besprochen. Hierfür wurde die Frage gestellt, auf welche Aspekte er bei dem Review der Releasevorbereitungen achtet. Basierend auf seiner Antwort wurde eine *Release Review Checkliste* zusammengestellt, die im Anhang D zu finden ist und zukünftigen Releaseteams als Leitfaden für die Prüfung ihrer Releasevorbereitungen dienen soll.

Das Befolgen von Checklisten hilft dabei, sich wiederholende, manuelle Tätigkeiten korrekt auszuführen und dabei nichts Essentielles zu vergessen (Beneken u. a. 2022, S.679f). Derartige Checklisten sollten bei jedem Durchlauf gepflegt, aktualisiert und korrigiert werden (Beneken u. a. 2022, S.680).

Zusätzlich wäre zu überlegen, den Reviewprozess der Releasevorbereitungen ähnlich wie die Qualitätssicherungsmaßnahmen auf den Hauptbranches vom `Renew`-GitLab-Projekt zu gestalten.

Würden im `renew-web`-GitLab-Projekt ein Merge-Request vom Prepare-Branch auf den `master`-Branch geöffnet werden, so könnten die durch Reviewende angemerkten Korrekturen und Nachfragen innerhalb des MR festgehalten und bearbeitet werden, anstatt verteilt in Mattermost-Nachrichten und Jira-Ticket-Kommentaren. Dadurch wäre der Reviewprozess für alle Beteiligten gebündelt einsehbar und auch für spätere Releaseteams innerhalb des geschlossenen MR sichtbar, sodass aus den Vorgehensweisen und Erkenntnissen früherer Releaseteams gelernt werden kann.

6.2.2 Versionierung der Plugins

Problembeschreibung

Bei der Veröffentlichung von RENEW 4.1 ist neben anderen Änderungen das Plugin *PTChannel* in die Menge der veröffentlichten Plugins aufgenommen worden.

Vor der Veröffentlichung wurde dessen Versionsnummer nicht überprüft, sodass es mit der Versionsnummer 0.1 veröffentlicht wurde, was nicht der Semantic-Versioning-Definition entspricht, welche bei der Versionierung von RENEWs Plugins befolgt werden soll. Es fehlt die laut Preston-Werner benötigte dritte Stelle, außerdem werden in der Definition

von Semantic Versioning Versionen mit einer führenden Null nicht als stabil angesehen (Preston-Werner 2013).

Vorschläge

Um zu verhindern, dass in Zukunft Versionierungsfehler wie beim neuen PTChannel-Plugin veröffentlicht werden, sollten vor einem Release die Plugin-Versionsnummern aller für das Release relevanten, veränderten Plugins überprüft werden.

Diese Prüfung ist bereits Teil einer während dieser Arbeit entworfenen Release Review Checkliste. Wenn eine solche Versionsnummer nicht schon erhöht wurde, sollte diese nun den Änderungen entsprechend in der MAJOR-, MINOR- oder PATCH-Stelle um eins erhöht werden.

Alternativ könnte das Problem früher im Entwicklungsprozess angegangen werden, indem die Erhöhung der Plugin-Versionsnummer Teil der Code Review Checkliste wird.

Allerdings wäre hier zu beachten, dass wenn mehrere Veränderungen an einem Plugin zwischen zwei Releases vorgenommen werden, die Nummer nicht mehrfach, sondern nur um eins erhöht wird.

6.2.3 Pluginbezeichnungen im Handbuch

Problembeschreibung

Die JAR-Dateinamen einiger Plugins von RENEW werden im Handbuch referenziert. Hierfür werden in den TEX-Dateien des Handbuchs Makros eingebunden, die vorher in der Datei `Core/doc/plugindefs.sty` definiert wurden.

Für die 4.x-Versionen von RENEW haben die JAR-Dateinamen der Plugins eine bestimmte Form, nämlich `<plugin-name>-MAJOR.MINOR.PATCH.jar`[3]. Hierbei ist darauf zu achten, dass die in den Makros in der `plugindefs.sty` angegebenen JAR-Dateinamen mit denen der tatsächlichen JARs in den Distribution-Packages übereinstimmen (s. 5.3 Aktualisierung der Dokumentation).

Innerhalb der von dieser Arbeit begleiteten Releases kam es vor, dass diese JAR-Dateinamen im Handbuch falsch waren. Zwar wurde der Fehler vor der Veröffentlichung durch den Review eines Hauptentwicklers entdeckt, jedoch hätte er dem Releaseteam schon früher auffallen sollen.

Vorschläge

Die Überprüfung der JAR-Dateinamen in der `plugindefs.sty` sollte zur Releasevorbereitung gehören.

Die erstellte Release Review Checkliste fragt diesen Punkt ab.

Alternativ könnte das Handbuch sorgfältig durchgegangen und nach Codezeilen gesucht werden, in denen die JAR-Dateinamen erwähnt werden, um diese mit den tatsächlichen abzugleichen. Durch diese Vorgehensweise würden auch Fehler gefunden, die aufgrund dessen auftreten, dass bei der Überarbeitung des Handbuchs an dieser Stelle keine Makros verwendet wurden.

[3] Für das Gui-Plugin ist das Makro bspw. folgendermaßen definiert: `\newcommand{\PluginGui}{de.renew.gui-2.0.0.jar}}`

Dieser Fall könnte auftreten, wenn die das Handbuch überarbeitende Person bspw. nicht wusste, dass Makros für diesen Zweck existieren und genutzt werden sollten.

Allerdings sollten die Hinweise auf die Verwendung der `plugindefs.sty` in der Dokumentation zur Aktualisierung des Handbuchs und in der Release Review Checkliste diesen unerwünschten Fall ausschließen.

Um den Prozess der Makro-Erstellung zuverlässiger zu gestalten, sollte für den `main`-Branch das Skript `prepare-plugin-versions.sh` angepasst und innerhalb der Releasevorbereitung eingesetzt werden, sodass die Makros für die JAR-Dateinahmen der Plugins automatisiert neu generiert werden und nicht mehr händisch aktualisiert werden müssen.

Eine Überprüfung innerhalb des Reviews sollte dennoch weiterhin durchgeführt werden, solange das Ausführen des Skripts nicht Teil der CI/CD-Pipeline ist, da nicht überprüft werden kann, ob das Skript tatsächlich ausgeführt wurde.

6.2.4 Lizenzbedingungen

Problembeschreibung

RENEWs Lizenzbedingungen sind in drei Abschnitte aufgeteilt: „Contributed Parts“, „Original Parts“ und „Disclaimer“.

In „Contributed Parts“ werden die Lizenzbedingungen von verwendeten Softwareteilen wiedergegeben, die von externen Personen entwickelt wurden und von RENEW verwendet werden. In „Original Parts“ werden die Lizenzbedingungen für die Softwareteile wiedergegeben, die nicht auf dem Werk anderer basieren. In „Disclaimer“ werden Ausschlussklauseln angegeben.

Die Lizenzbedingungen sind sowohl in der TEX-Datei `license.tex` als auch in der HTML-Datei `license.html` festgehalten.

Die Lizenzbedingungen müssen selten grundlegend aktualisiert werden. Durch diesen Umstand haben die Lizenzbedingungen während der letzten Releases wenig Beachtung erhalten und aus diesem Grund sind einige Fehler aufgetreten.

Auf der Webseite wurde die Aktualisierung des Copyright-Datums im Abschnitt „Original Parts“ versäumt, weil sie bis zur Erstellung dieser Arbeit nirgendwo dokumentiert wurde und das Versäumnis keiner der an den Releasevorbereitung beteiligten Personen aufgefallen ist.

Bei den begleiteten Releases wurde außerdem vernachlässigt, die Lizenzbedingungen im Handbuch und auf der Webseite gleichermaßen zu pflegen, sodass diese nach der Veröffentlichung von RENEW 4.1 voneinander abweichen.

Während der Erstellung dieser Arbeit wurden die Differenzen gesammelt und es wurde determiniert, an welchen Stellen die TEX- und an welchen die HTML-Datei aktueller ist. Das Ergebnis wurde an das Releaseteam, welches das Release von Version 4.2 vorbereitet, kommuniziert, sodass beide Dateien nach der nächsten Veröffentlichung wieder aktuell sind und ihre Inhalte miteinander übereinstimmen.

Vorschlag

Innerhalb der Releasevorbereitungen sollten wenigstens drei Fragen gestellt werden:

1. Wurde das Copyright-Datum im Abschnitt „Original Parts“ aktualisiert[4]?
2. Wurden für Neuerungen im bevorstehenden Release externe Techniken verwendet, die im Abschnitt „Contributed Parts“ angegeben werden müssen?
3. Wurden etwaige Änderungen an den Lizenzbedingungen sowohl in der TEX- als auch in der HTML-Datei vorgenommen?

Alle drei Punkte wurden in der Dokumentation der Durchführung eines Release festgehalten, sodass, wenn diese bei künftigen Releases verwendet wird, die Lizenzbedingungen aktualisiert werden und konsistent in Handbuch und auf der Webseite vorliegen sollten.

Die folgenden Überlegungen sollten für den zukünftigen Umgang mit den Lizenzbedingungen erwogen werden.

Punkt 2 sollte in die Code Review Checkliste aufgenommen werden, denn die Entwickelnden, die Neuerungen zu RENEW beisteuern, sollten i.d.R. besser beurteilen können, ob bzw. welche externe Arbeit die von ihnen entwickelten Funktionalitäten nutzen.

Wenn die Pflege der Lizenzbedingungen im Handbuch Teil der Code Review Checkliste werden würde, so müsste das Releaseteam diese später lediglich auf die Webseite übertragen. Des Weiteren würde dieser nicht zu automatisierende Schritt von der Releasevorbereitung in die Entwicklung bzw. die QA der Entwicklung verlegt werden, sodass der vollständigen Automatisierung des Releaseprozesses ein Hindernis weniger im Weg stehen würde.

Während sich Punkt 2 nicht automatisieren lässt und lediglich durch Konventionen festgelegt werden sollte, wann die Lizenzbedingungen auf Vollständigkeit überprüft werden sollen, lassen sich die Punkte 1 und 3 u.U. automatisieren.

Punkt 1 ließe sich mit einem einfachen Skript automatisieren, welches das Copyright-Datum auf das aktuelle Jahr aktualisiert.

Dafür könnte Screen Scraping bspw. durch Beautiful Soup[5] eingesetzt werden oder die neue Generierung der `license.html` mit Einsatz eines Mustache-Templates[6] angestrebt werden.

Die Ausführung des neu erstellten Skripts müsste Teil der Dokumentation, wie die Webseite für ein neues Release vorbereitet wird, werden.

Punkt 3 ist wichtig, zeigt jedoch auch die Redundanz darin auf, dass die Lizenzbedingungen an zwei völlig unterschiedlichen Stellen synchron gepflegt werden muss.

Es sollte überlegt werden, wie die `license.html` aus der `license.tex` generiert werden kann. Hierfür könnte es sinnvoll sein, die `license.tex` in der aus dem RENEW-GitLab-Projekt generierten `packages.zip`-Datei zu inkludieren, sodass aus dieser durch eine entsprechende Erweiterung des Skripts `createfromtemplates.py` die `license.html` generiert wird.

6.2.5 Redundante Dokumentation

Problembeschreibung

Die Dokumentation von RENEW ist redundant. Es existieren die TEX-Dateien, aus denen das Handbuch generiert wird, und insgesamt drei Readme-Dateien für RENEW, von denen zwei

[4] Für das Handbuch passiert das automatisiert bei der Generierung aus den TEX-Dateien.

[5] Dokumentation von Beautiful Soup: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`

[6] Mustache-Templates werden auch für die Generierung der versionsspezifischen HTML-Dateien verwendet (s. 5.4.2 Generierung eines Versionsordners).

veröffentlicht werden.

Das Handbuch und die Readme-Dateien sind außerdem recht fehleranfällig. Die Dokumentation ist sehr umfangreich und für neue Entwickelnde, die an den Releasevorbereitungen beteiligt sind, unübersichtlich. Die händisch vorzunehmenden Änderungen sind nicht anspruchsvoll, aber bedürfen Konzentration. Beim Testen eines Release liegt der Fokus des Releaseteams nicht zwingend auf der Überprüfung der Dokumentation.

So steht bspw. in der `README` des Release 4.1 die alte Versionsnummer 4.0.

Vorschläge

Es wäre ratsam, die unterschiedlichen Dokumentationen zu konsolidieren, sodass bei einem Release Dinge wie Versionsnummern, Veröffentlichungsdaten und benötigte Java-Versionen nicht mehrfach aktualisiert werden müssen.

Dies würde die Fehleranfälligkeit bereits etwas reduzieren.

Alternativ könnte angestrebt werden, die unterschiedlichen Dokumentationen aus einer Quelle zu generieren, um Redundanzen und den damit verbundenen Pflegeaufwand zu reduzieren.

Die `LICENSE`-Datei, die Teil des Release ist, wird bspw. aus den TEX-Dateien generiert. Möglichkeiten, auch die Readme-Dateien aus TEX-Dateien zu generieren, sollten geprüft werden.

Außerdem wäre zu überlegen, ob die aktuelle Versionsnummer und die benötigte Java-Version automatisch aus der `release-tasks.gradle` bzw. der `build.gradle` in die Dokumentation übernommen werden können.

Auf diese Art würde die Anzahl der redundant händisch auszuführenden Schritte bei der Aktualisierung der Dokumentation für ein neues Release reduziert werden.

6.2.6 Veraltete Dokumentation

Problembeschreibung

Zusätzlich zu den TEX-Dateien, aus denen RENEWs Handbuch generiert wird, gibt es weitere TEX-Dateien, aus denen ein Architecture-Guide generiert werden kann, welcher allerdings sehr veraltet ist und die Architektur von RENEW 1.6 beschreibt.

Während der Beschäftigung mit den jüngsten drei Releases für diese Arbeit fiel auf, dass dieser Architecture-Guide – sowohl dessen TEX-Dateien, als auch die generierte PDF-Datei – in den veröffentlichten Source-Packages enthalten ist.

Außerdem wird auf den Architecture-Guide in der `README` unter „Further Reading“ hingewiesen, allerdings fehlt die Erwähnung, dass dieser nur im Source-Package zu finden und nicht aktuell ist.

Vorschläge

Wenn weiterhin ein Architecture-Guide veröffentlicht werden soll, muss dieser aktuell sein.

Das bedeutet auch, dass die Aktualisierung der entsprechenden Dateien in den Releaseprozess bzw. in den Entwicklungsprozess aufgenommen werden müsste und die Überprüfung dieser in die Release Review Checkliste bzw. in die Code Review Checkliste.

Sollte dies nicht geschehen, wäre es sinnvoll, den Architecture-Guide bis auf Weiteres nicht mehr zu veröffentlichen.

Dafür müsste die `release-tasks.gradle` angepasst und der Hinweis in der `README` entfernt werden.

Darüber hinaus sollten in diesem Fall die Dateien des Architecture-Guides im `Renew-GitLab`-Projekt gelöscht werden, denn Dokumentation sollte – so wie Code auch – sauber gehalten werden (Google LLC 2015).

6.3 Organisation

Gewisse Fragestellungen innerhalb des Releaseprozesses von RENEW betreffen die Organisation desselben.

Im Folgenden werden fehlende Codefreeze-Zeiten, die Durchführung von Hotfix-Releases, die Trennung von Entwicklung und Release, die Wartung mehrerer MAJOR-Versionen, die Beseitigung von Bugs und der Wissenserhalt bzgl. der Releases innerhalb der Entwicklung und Veröffentlichung von RENEW betrachtet.

6.3.1 Codefreeze

Problembeschreibung

RENEWs Release-Management-Strategie lässt sich am ehesten als zeitbasierte Entwicklung bezeichnen. Das bedeutet, dass ein bestimmtes Datum als Releasezeitpunkt festgelegt wird und weniger die Entwicklung bestimmter neuer Funktionen über die Entscheidung, ein neues Release zu veröffentlichen, bestimmt.

Allerdings war das jeweils angestrebte Veröffentlichungsdatum nicht so fest, dass das Einhalten des Datums höher als die Qualität des jeweiligen Release priorisiert wurde.

Diese Release-Management-Strategie hatte bei den von dieser Arbeit begleiteten Releases zur Folge, dass noch während der laufenden Releasevorbereitungen Neuerungen in das anstehende Release aufgenommen wurden. Dieser Umstand bedeutete, dass das Releaseteam bestimmte Schritte, wie das Erstellen des Changelogs und das Testen der zu veröffentlichenden Version, mehrfach durchführen musste.

Dieses Vorgehen ist zum einen nicht sehr effizient und zum anderen entstanden so vermeidbare Fehler, da der Überblick über die zu wiederholenden Tätigkeiten z.T. verloren ging. Hierdurch wichen bspw. Informationen im Handbuch und auf der Webseite zum Zeitpunkt eines Release voneinander ab.

Zusätzlich zu den Effizienzeinbußen und Fehlern bei der Releasevorbereitung bedeutet das Aufnehmen von weniger getesteten Neuerungen auch, dass das Vertrauen in die Gewährleistung der Qualität der veröffentlichten Software sinkt. Die verschiedenen Distribution-Packages müssen auf verschiedenen Systemen getestet werden und Zeitdruck beim Testen führt zum Übersehen von fehlerhaftem Verhalten der Software.

Vorschläge

Um die Releasevorbereitungen effizienter durchführen zu können wäre es sinnvoll, entweder Codefreeze-Zeiten einzuführen (Michlmayr u. a. 2007) oder einen Release-Branch zu erstellen, in den während der Vorbereitungen nur noch Bugfixes u.ä. integriert werden, um diesen zu stabilisieren (Preißel und Stachmann 2019, Kap.20).

Auf diese Art könnte eine Phase des parallelen Debuggens bzw. der Stabilisierung stärker betont werden, sodass die zu veröffentlichende Version robuster getestet wird. Außerdem könnte das Releaseteam so einen besseren Überblick darüber behalten, ob nachträglich, also nach Generierung der Artefakte für die Veröffentlichung, noch release-relevante Änderungen vorgenommen wurden.

Karvonen u. a. identifizieren das Synchronisieren aller Veröffentlichungszeitpläne, -aktivitäten und -übergaben zwischen allen internen Stakeholdern, also Entwickelnden, Testenden und Produktmanagenden, als eine Voraussetzung für Continuous Deployment (Karvonen u. a. 2017).

Wenn RENEWs Veröffentlichungsprozess in diese Richtung gehen soll, wäre also eine klarere Kommunikation und Abstimmung zwischen Releaseteam, CPO bzw. CCPO und den anderen Entwicklungsteams notwendig.

6.3.2 Hotfix-Release

Problembeschreibung

Hotfixes sind dringliche Änderungen, die Sicherheitsrisiken oder schwere Fehler in einer veröffentlichten Software beheben. Sie sollten unabhängig von anderen Änderungen so schnell wie möglich veröffentlicht werden (Preißel und Stachmann 2019, Kap.20).

In der Veröffentlichungspraxis von RENEW sind solche Hotfixes nicht ohne Weiteres möglich.

Sollte für RENEW ein Hotfix veröffentlicht werden müssen, so wäre es nötig, zunächst den Commit zu finden, welcher den für die Generierung der veröffentlichten Artefakte verwendeten Code markiert. Dies ist mit Hilfe der Release-Tags möglich, welche nach jeder Veröffentlichung von einem Hauptentwickler händisch gesetzt werden.

Dieser Commit müsste nun ausgecheckt, der Hotfix implementiert und die Artefakte für das Hotfix-Release generiert werden, sodass die Veröffentlichung des Hotfixes tatsächlich nur diesen enthält und nicht etwaige andere Änderungen, die bereits auf den Hauptbranch gepusht wurden und bedingen würden, dass ein vollwertiges Release anstelle eines Hotfix-Release durchgeführt werden müsste.

Danach müsste der Hotfix natürlich auch in allen anderen Branches übernommen werden, damit er in folgenden Releases bereits vorhanden ist.

Vorschlag

Preißel und Stachmann schlagen vor, eine Branching-Strategie anzuwenden, bei der auf einem `develop`-Branch Neuerungen entwickelt werden, auf einem `release`-Branch ein bevorstehendes Release stabilisiert wird, auf dem `master`-Branch das aktuelle Release liegt und in `hotfix`-Branches Fehler im `master` behoben werden.

Der `release`-Branch und die `hotfix`-Branches existieren dabei nur, solange die Stabilisierung bzw. die Fehlerbehebung durchgeführt wird (Preißel und Stachmann 2019, Kap.20). Änderungen, die in diesen Branches vorgenommen wurden, werden immer durch Merges in den `develop`-Branch zurückgeführt (Preißel und Stachmann 2019, Kap.20).

Diese Branching-Strategie ist ähnlich zu Git-Flow[7] (Preißel und Stachmann 2019).

[7] Git-Flow ist von Vincent Driessen entworfene Branching-Strategie, die er in einem Blog-Post beschrieb: `https://nvie.com/posts/a-successful-git-branching-model/`

Für periodisch bzw. nicht kontinuierlich durchgeführte Releases, wie sie bei RENEW derzeit durchgeführt werden, sollte überlegt werden, eine solche Branching-Strategie zu verfolgen.

Der Vorteil gegenüber dem bloßen Einsatz von Tags wäre, dass die Git-Historie der Releases und Hotfix-Releases besser nachzuvollziehen wäre (Preißel und Stachmann 2019, Kap.20).

Ein Hotfix-Release würde allgemein weniger Vorbereitung bedürfen als ein reguläres Release, denn es müsste nicht so aufwändig getestet werden und es müssten keine neuen Features dokumentiert werden.

Bei der RENEW-Versionsnummer müsste natürlich die PATCH-Versionsnummer erhöht werden und die Dokumentation und die Webseite müssten trotzdem bezüglich der Versionsnummer, des -datums, der Historie, der „Known Issues“ und des Archivs der alten Versionen angepasst werden (s. 5.3 Aktualisierung der Dokumentation und 5.4.3 Händische Anpassung von HTML-Dateien).

6.3.3 Trennung von Entwicklung und Release

Problembeschreibung

Die in Abschnitt 6.3.2 bereits beschriebene Branching-Strategie würde eine weitere bisherige Herausforderung innerhalb des Veröffentlichungsprozesses von RENEW adressieren, nämlich den Umstand, dass es bisher keine Trennung auf Branch-Ebene zwischen Entwicklung und Release existiert.

In den Grundlagen wurde ein möglicher OSS-Entwicklungs-Lebenszyklus beschrieben. Dieser beinhaltet die Phasen *development release* und *production release*, in denen Änderungen jeweils in den entsprechenden Branch integriert werden.

In der Entwicklung von RENEW entsprechen sowohl der Development-Branch als auch der Release-Branch dem Hauptbranch, auf dem theoretisch alles production-ready sein sollte und auf dem während der Releasevorbereitung etwaige Stabilisierungen vor der Veröffentlichung vorgenommen werden. Diese Anpassungen vor der Veröffentlichung geschehen dabei parallel zur regulären Entwicklung neuer Features.

Dies führt zu Unübersichtlichkeit z.B. im Hinblick darauf, welche Merge-Requests noch vor der Veröffentlichung geschlossen werden müssen und welche für das anstehende Release irrelevant sind.

Vorschlag

Durch die Verwendung von `development-` und `release`-Branches (s. 6.3.2 Hotfix-Release) wäre eine saubere, organisatorische Trennung zwischen Stabilisierung des Release und weiterer Entwicklung möglich. Das Releaseteam könnte auf einen Blick sehen, ob noch Merge-Requests offen sind, die den `release`-Branch als Zielbranch haben, anstatt wie bisher alle offenen Merge-Requests händisch danach zu durchsuchen, ob sie veröffentlichte Plugins betreffen und die Änderungen in das aktuell vorzubereitende Release aufgenommen werden sollen. Dies würde die Wahrscheinlichkeit, relevante Merge-Requests zu übersehen, reduzieren.

6.3.4 Wartung mehrerer MAJOR-Versionen

Problembeschreibung

Zur Zeit werden zwei MAJOR-Versionen von RENEW entwickelt und gewartet – nämlich die klassische 2.x-Variante und die modulare 4.x-Variante (Kummer u. a. 2022; Kummer u. a. 2023a).

Zusätzlich ist eine weitere MAJOR-Version 5.0 bereits in Planung (Heinze 2022). Mit der 5.0-Variante würde RENEW auf mehrere Repositories aufgeteilt werden, sodass jedes Plugin ein eigenes Repository besitzt und die Plugins sich gegenseitig als Maven-Projekte einbinden (Heinze 2022). Hierdurch würde sich auch der Releaseprozess deutlich verändern (Heinze 2022).

Vorschlag

Das Pflegen mehrerer, aktueller Versionen wirkt sich negativ auf den Wartungsaufwand aus (Aggarwal und Mani 2019). Aus diesem Grund sollte angestrebt werden, nur noch eine MAJOR-Version aktiv zu entwickeln.

Um dieses Ziel zu erreichen sollte die Modularisierung von Plugins, die bisher lediglich in der klassischen Variante von RENEW enthalten sind, vorangetrieben werden, sodass die modulare Variante die klassische vollständig ablösen kann. Danach könnte eine Umsetzung der von Heinze vorgeschlagenen Aufteilung des RENEW-Projekts erfolgen, nach welcher nur noch diese weiterentwickelt wird.

Sollten weiterhin mehrere MAJOR-Versionen entwickelt und gepflegt werden sollen, so muss der doppelte bzw. dreifache Entwicklungs- und Wartungsaufwand hingenommen werden.

6.3.5 Beseitigung von Bugs

Problembeschreibung

Eine mögliche Metrik für die Bemessung der Qualität eines Release ist die Anzahl der offenen Bugtickets (Adams u. a. 2015). Diese Metrik kann auch mit einer Obergrenze (engl.: defect limit) belegt werden, sodass die regulären Entwicklungstätigkeiten unterbrochen werden müssen, sobald sie überschritten wurde (Aggarwal und Mani 2019).

Während der von dieser Arbeit begleiteten RENEW-Veröffentlichungen wurde kein koordinierter Aufwand beobachtet, offene Bugtickets zu schließen oder die „Known Issues“ zu adressieren.

Bugs wurden eher zufällig oder durch erfahrenere Entwickelnde entdeckt und hoffentlich noch vor dem Release behoben, ansonsten wurde es mit weiteren „Known Issues“ veröffentlicht. RENEW 4.0 wurde mit denselben „Known Issues“ veröffentlicht wie 4.1.

Alle dokumentierten Bugs sind im Ticketsystem Jira mit den entsprechenden Filtern über alle Unterprojekte von RENEW hinweg einsehbar[8].

Dabei fehlt die Einordnung, ob die Bugs innerhalb von veröffentlichten Plugins auftreten. Dies wäre eine wichtige Information, um die Anzahl der offenen Bugtickets als Metrik für die Bemessung der Releasequalität nutzen zu können.

[8] https://tgipm.informatik.uni-hamburg.de/jira/browse/GUI-93?jql=resolution%20%3D%20Unresolved%20AND%20issuetype%20%3D%20Bug%20ORDER%20BY%20updated%20DESC

Vorschlag

Ein Defect Backlog, also eine Liste der bekannten Bugs und „Known Issues“, sollte für alle an der Entwicklung beteiligten Personen klar sichtbar sein und Teile des Entwicklungsteams sollten verantwortlich dafür sein, dieses Backlog zu reduzieren (Humble und Farley 2011, Kap.4).

Offene Bugtickets sollten während der Releasevorbereitung mehr Aufmerksamkeit bekommen in der Form, dass die Anzahl der offenen Bugtickets, die die veröffentlichten Plugins betreffen, erhoben wird und in Rücksprache mit CPO und CCPO vereinbart wird, wie hoch diese Anzahl höchstens sein darf, damit ein Release veröffentlicht werden kann.

Mindestens das Releaseteam, besser das gesamte Entwicklungsteam, sollte vor einem Release den Fokus auf die Behebung von Bugs legen, um das Release zu stabilisieren (s. Abschnitt 6.3.1).

Allgemein sollten Bugtickets nach Möglichkeit nicht für die Teilnehmenden zukünftiger RENEW-Lehrprojekte offen gelassen werden, da so die Gefahr größer wird, dass sie gänzlich in Vergessenheit geraten oder das benötigte Wissen für die Behebung verloren geht (Humble und Farley 2011, Kap.4).

6.3.6 Wissenserhalt bzgl. Veröffentlichungen

Problembeschreibung

Durch den universitären Kontext, in dem sich RENEWs Entwicklung und Veröffentlichung bewegen, besteht der Umstand, dass der Kreis der Entwickelnden einer recht hohen Fluktuation ausgesetzt ist. Die Personen, die an der Entwicklung teilnehmen, tun dies mitunter nur für ein Semester und selbst wenn sie länger dabei bleiben, werden sie spätestens nach dem Erlangen des von ihnen angestrebten universitären Abschlusses aus der Entwicklung ausscheiden.

Dieser Umstand resultiert in der Herausforderung, dass projekt-spezifisches Wissen, Erfahrungen und Expertise ständig für die fortlaufende Entwicklung verloren gehen. Diese Herausforderung besteht in vielen OSS Projekten, bei denen die Personengruppe der an der Entwicklung Beteiligten ebenfalls fluktuiert (Rashid u. a. 2019).

Um dem Verlust des gesammelten und erarbeiteten Wissens entgegenzuwirken, muss es ständig weitervermittelt werden.

Innerhalb der RENEW-Entwicklung wird Wissen in Abschlussarbeiten, Projektberichten, veröffentlichten Papern und auf internen Confluence-Seiten aufbereitet. Zusätzlich wird Grundlegendes in der Onboarding-Phase der Lehrprojekte vermittelt (Anonymous Author(s) 2024). Nicht dokumentiertes Wissen wird über die PO und SM der einzelnen Teams weitervermittelt, welche wiederum CPO, SoSM, CCPO oder das Organisationsteam des gesamten Projekts bei Fragen konsultieren.

So funktioniert die Weitervermittlung von Wissen und die Hoffnung ist, dass dabei möglichst wenig Zeit für unnötige Recherchen zu bereits behandelten Themen verwendet wird.

Eine Herausforderung hierbei ist allerdings, dass es relativ viele Lücken in der Dokumentation des Wissens gibt.

Studierende sind zwar dazu angehalten, ihr Vorgehen auf Confluence-Seiten und in Projektberichten festzuhalten, allerdings passiert dies nicht stringent genug, wodurch große Lücken in der Dokumentation bleiben. Des Weiteren sind die Studierenden frei in ihrer Themenwahl für Projektberichte, wodurch nicht sichergestellt ist, dass die Auswahl der Themen das gesamte behandelte Themenspektrum abdeckt.

Abschlussarbeitende wiederum haben ein spezielles Thema, auf das sie sich fokussieren und welches sie verschriftlichen, hierbei werden aber u.U. nicht alle grundlegenden Aspekte festgehalten, da diese möglicherweise als selbstverständlich vorausgesetzt werden.

Bezogen auf den Veröffentlichungsprozess wird von den Studierenden allgemein tendenziell weniger geschrieben, da die Schwerpunkte der Arbeiten i.d.R. auf dem Entwickeln neuer Plugins, Instandhaltung der Software oder anderen DevOps-Themen als der letztendlichen Veröffentlichung liegen.

Vorschläge

Da durch diese Arbeit der Veröffentlichungsprozess mehrerer RENEW-Releases begleitet und dokumentiert wurde, sollten Wissenslücken in diesem Themenbereich für zukünftige Releaseteams zu schließen sein.

Basierend auf den Kapiteln 4 und 5 wird bereits ein Release von RENEW 4.2 vorbereitet. Das teaminterne Review der Releasevorbereitungen wird mit der erarbeiteten Release Review Checkliste durchgeführt werden.

Da sich der Releaseprozess in Zukunft verändern wird, sollten die Dokumentation der in dieser Arbeit festgehaltenen Arbeitsweisen und Konventionen fortlaufend aktualisiert werden.

Michlmayr u. a. beschreiben als eine Praktik des Release-Managements das Abhalten eines Reviews nach der Veröffentlichung (Michlmayr u. a. 2007).

Die Einführung eines solchen Reviews zu den Releasevorbereitungen und der Durchführung des Release könnte innerhalb der Entwicklung von RENEW sinnvoll sein, um sowohl die Dokumentation der Durchführung als auch die Release Review Checkliste zu aktualisieren und ggf. auszubauen, denn innerhalb jeder Veröffentlichung werden Erfahrungen und Erkenntnisse durch die Personen des jeweils ausführenden Releaseteams gesammelt und der Prozess weiterentwickelt (Beneken u. a. 2022, S.680).

6.4 Automatisierung

In diesem Abschnitt werden Automatisierungsmöglichkeiten innerhalb RENEWs Veröffentlichungsprozesses beschrieben, im Genaueren bei der Erstellung des Changelogs, der Umsetzung von CD, dem Bereitstellen von Nightly Builds und der Aktualisierung der Webseiten.

6.4.1 Erstellung des Changelogs

Problembeschreibung

Zu den Releasevorbereitungen gehört die Erstellung eines Changelogs, welches die wichtigsten Änderungen im Vergleich zum vorigen Release zusammenfasst. Das Changelog wird sowohl im Handbuch, als auch auf der Webseite von RENEW als Release History wiedergegeben.

Die Erstellung des Changelogs für RENEW-Releases erfolgt bisher händisch und durch Sichtung der Commits, die seit der Erstellung des letzten Release auf dem jeweiligen Hauptbranch (`master` oder `main`) getätigt wurden.

Es wäre erstrebenswert, die Changelogerstellung als Teil des Releaseprozesses zu automatisieren, da diese Aufgabe zeitaufwändig und anspruchsvoll ist (Jiang u. a. 2022; Nath und Roy 2021).

Vorschläge

Commits bilden die Grundlage für das Erstellen des Changelogs.

Die simpelste Möglichkeit, die Changelogerstellung zu automatisieren, wäre, die Nachrichten aller Commits, die seit der Veröffentlichung des letzten Release auf dem Hauptbranch getätigt wurden, automatisiert zu sammeln und in einem Ausgabedokument aufzulisten.

Dieser Ansatz ist allerdings nicht erstrebenswert, da in der Commit-Historie viele Informationen vorhanden sind, die nicht in ein übersichtliches Changelog gehören, welches nur die relevantesten Änderungen beschreibt.

Wenn das Changelog eine Zusammenfassung der wichtigsten Änderungen bleiben soll, ist es also nötig, entweder die Quelle der Informationen, also die Commit-Nachrichten, anzupassen, diese Informationen so zu filtern, dass nur die wichtigsten in das Changelog aufgenommen werden, oder eine Kombination aus diesen Optionen anzuwenden.

Im `Renew`-Projekt werden bereits Commit-Nachrichten auf die Einhaltung einer festgelegten Konvention[9] durch Git Hooks[10] überprüft.

Es wäre möglich mit Git Hooks zusätzlich sicherzustellen, dass die Commit-Nachrichten der Commits, die auf dem Hauptbranch ausgeführt werden, eine Information darüber enthalten, welchem Typ der Commit angehört – z.B. angelehnt an Conventional Commits[11].

Bei der automatischen Changelogerstellung könnte daraufhin angegeben werden, welche Typen von Commits berücksichtigt werden sollen (Hoffmann 2020).

Darüber hinaus existieren Möglichkeiten, ein übersichtliches Changelog gestützt durch maschinelles Lernen aus Commit-Nachrichten zu generieren (Jiang u. a. 2022).

6.4.2 Umsetzung von CD

Problembeschreibung

Innerhalb des Entwicklungs- bzw. Veröffentlichungsprozesses von RENEW wird bisher CI insofern umgesetzt, als dass Commits in das Remote-Repository eine Pipeline der GitLab CI/CD auslösen. Der so eingecheckte Code wird innerhalb dieser Pipeline gebaut und getestet. Wie bereits erwähnt, ist die Testabdeckung hierbei allerdings nicht umfangreich genug (s. 6.1.1 Testabdeckung).

Des Weiteren erfolgt bisher keine Umsetzung von CD – weder von Continuous Delivery, bei dem die Software nach dem erfolgreichen Durchlaufen der CI und Akzeptanztests automatisiert in eine Testumgebung ausgeliefert wird, noch von Continuous Deployment, bei dem die Software nach erfolgreicher Delivery zusätzlich automatisiert in die Produktionsumgebung ausgeliefert wird.

[9] Confluence-Dokumentation des Commit-Message-Templates: `https://tgipm.informatik.uni-hamburg.de/confluence/display/RENEWDEV/Git#Git-CommitMessages`

[10] Git Hooks-Dokumentation: `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`

[11] Conventional Commits: `https://www.conventionalcommits.org/en/v1.0.0/`

Vorschläge

In der Literatur wird CD häufig für Softwareanwendungen beschrieben, die SaaS[12] sind (s. Bryant und Marín-Pérez 2018; Humble und Farley 2011), also Anwendungen, die auf einem Server laufen und von Nutzenden über einen Webbrowser aufgerufen werden (Rowell 2023). RENEW ist im Gegensatz dazu eine Non-SaaS, was bedeutet, dass Nutzende die Software selbst auf ihren Geräten installieren, um sie zu verwenden (Rowell 2023).

Die Produktionsumgebung ist demnach nicht zentral verwaltbar, allerdings kann die Bereitstellung der RENEW-Releases auf der öffentlichen Webseite als Auslieferung der Software angesehen werden.

Unabhängig von der Art der Auslieferung, bleiben die zentralen Konzepte, Grundsätze und Behauptungen zur Umsetzung von CD unverändert (Bryant und Marín-Pérez 2018, S.21).

Tatsächliches Continuous Deployment für eine Desktopanwendung wie RENEW würde bedeuten, den Upgradeprozess von installierten RENEW-Instanzen auf die neueste, veröffentlichte Version automatisiert durchzuführen.

Hierfür müsste im Hinblick auf den Upgradeprozess überlegt werden, wie dieser gemanagt werden kann, wie bestehende Binärdateien, Daten und Konfigurationen migriert werden können, wie der Prozess getestet werden kann und wie Absturz- und Fehlerprotokolle zur Laufzeit erlangt werden können (Humble und Farley 2011, S.267ff).

Da die Umsetzung automatischer Upgrades für RENEW den bisherigen Veröffentlichungsprozess vollkommen verändern würde und den Umfang dieser Arbeit weit übersteigt, da die Umsetzung keineswegs trivial oder eindeutig ist (Fitz 2009), wird diese Option hier nicht weiter beleuchtet.

Für den aktuellen Veröffentlichungsprozess von RENEW könnte eine Art Umsetzung von Continuous Delivery so aussehen, dass die Bereitstellung eines automatisch generierten Pre-Release inklusive der Unterwebseiten und der Release-Artefakte einer neuen Version zu Test- bzw. Reviewzwecken auf einer dafür eingerichteten Webseite erfolgt. Die Webseite entspräche der Testumgebung und sollte nicht öffentlich zugänglich sein, sondern nur für RENEW-Entwickelnde.

Außerdem sollte untersucht werden, welche Akzeptanztests innerhalb einer CD-Pipeline für RENEW sinnvoll wären. Bevor automatisierte Akzeptanztests für RENEW existieren, die das Erstellen von fehlerhaften Pre-Releases verhindern, sollte für die Generierung abgewogen werden, in welchem Zeitintervall bzw. durch welche Events sie ausgelöst wird; bspw. wöchentlich oder durch Merges in den Hauptbranch.

Erst wenn auf diese Weise Continuous Delivery erfolgreich für RENEW implementiert wurde, kann darüber nachgedacht werden, ob sie auf eine Art Continuous Deployment ausgeweitet werden kann und neue Versionen von RENEW ständig und vollautomatisiert auf der öffentlichen Webseite veröffentlicht werden können. Jedoch ist es bei vielen Systemen erstrebenswert, manuelle Tests vor der Veröffentlichung durchzuführen (Humble und Farley 2011, S.126).

[12] **S**oftware **a**s **a** **S**ervice

6.4.3 Erstellung von Nightly Builds

Problembeschreibung

Innerhalb von RENEWs Veröffentlichungen gibt es bisher keine Nightly Builds. Diese sind jedoch vom CCPO des Softwareprojekts gewünscht.

Nightly Builds können dafür genutzt werden, den aktuellen Stand der Software auf Fehler zu testen, sicherzugehen, dass die Build- und Deploy-Pipelines noch richtig funktionieren, und, wenn sie veröffentlicht werden, Nutzenden die neueste, potenziell instabile Version der Software zur Verfügung zu stellen.

Vorschläge

Um Nightly Builds von RENEW zu ermöglichen, müsste zunächst der gesamte Veröffentlichungsprozess automatisiert werden, sodass eine Deployment-Pipeline besteht, die zu diesem Zweck genutzt werden kann.

Des Weiteren müsste ein Server aufgesetzt werden, auf den die automatisch generierten Artefakte für die Veröffentlichung automatisiert gepusht werden können.

Es wäre sinnvoll für die Nightly Builds einen eigenen Branch im `renew-web`-Projekt einzurichten.

Das Deployment könnte dann ähnlich wie das für die regulären Releases erfolgen (s. 5.6.1 Deployment auf TGI-Server).

Außerdem sind die Veröffentlichungsintervalle und die Benennung der Nightly Builds festzulegen. Gegen zu kurze, reguläre Intervalle spräche, dass mitunter keine Neuerungen innerhalb der release-relevanten Plugins während eines festgelegten Zeitabschnitts erfolgt sind (Silva u. a. 2016).

Eine derartige Veröffentlichung sollte zunächst nur intern auf einem Server des Fachbereichs erfolgen, damit die Deployment-Pipeline, die Veröffentlichungsintervalle der Nightly Builds und die Qualität der Nightly Builds evaluiert werden können, bevor sie der Allgemeinheit zugänglich gemacht werden.

Wenn die Veröffentlichung nur intern erfolgt, kann überlegt werden, inwiefern die Erstellung der Dokumentation und die Aktualisierung der Webseiten für die Nightly Build-Releases ausgelassen oder angepasst werden kann. Bisher enthalten diese Vorgänge viele händische Arbeitsschritte und es wäre zu überlegen, inwiefern für interne Nightly Builds bspw. ein aufbereitetes Changelog durch ein automatisch aus den Commit-Nachrichten generiertes ersetzt werden kann.

Zum einen sollte der Veöffentlichungsprozess der Nightly Builds möglichst ähnlich zum tatsächlichen Veröffentlichungsprozess sein, damit die Automatisierungen in Zukunft für den regulären Veröffentlichungsprozess genutzt werden können, zum anderen sollte der Aufbau der vollständig automatisierten Deployment-Pipeline iterativ erfolgen, sodass zunächst die unerlässlichen Arbeitsschritte automatisiert werden (Humble und Farley 2011, S.25f).

Die Einführung von Nightly Builds innerhalb von RENEWs Entwicklung ist das Thema einer bereits geplanten anderen Abschlussarbeit.

6.4.4 Aktualisierung der Webseiten

Problembeschreibung

Bisher wird für ein bevorstehendes Release ein `prepare`-Branch händisch im `renew-web`-Projekt angelegt, auf dem manuell eine neue JSON-Kontextdatei angelegt werden muss, die im `Renew`-Projekt generierten Artefakte händisch entpackt werden müssen und später manuelle Änderungen an den HTML-Dateien vorgenommen werden müssen (s. 5.4 Vorbereitung der Webseiten).

Während der von dieser Arbeit begleiteten Releases wurde dabei Folgendes festgestellt: Durch Änderungen am Handbuch oder am Code von RENEW nach der so aktualisierten Webseite, mussten gewisse Schritte mehrfach durchgeführt werden, wodurch Fehler bei den Dateigrößenangaben auf der Startseite eingebaut wurden, Abweichungen zwischen Handbuch und Webseite entstanden und generell mehr Zeit für eigentlich einmalig auszuführende Schritte aufgewendet werden musste.

Um diese menschlichen Fehler zu minimieren und Effizienz zu steigern, sollten die Arbeitsschritte für die Aktualisierung der Webseite automatisiert werden.

Vorschläge

Während des Lehrprojekts AOSE22 im Wintersemester 2022/23 wurden von Projekteilnehmern, die auch Mitglieder der Releaseteams waren, bereits Vorschläge erarbeitet, wie Teile der Aktualisierung der Webseiten im `renew-web`-Projekt automatisiert werden können.

Hierfür wurde die Möglichkeit untersucht, wie für `renew-web` eine CI/CD-Pipeline eingerichtet werden kann, innerhalb der die für ein Release benötigten Artefakte aus der `Renew`-CI/CD-Pipeline als Upstream-Pipeline gezogen werden, ein neuer `prepare`-Branch automatisch angelegt wird, eine neue JSON-Kontextdatei generiert wird, das Skript für die Erzeugung eines neuen Versionsordners ausgeführt wird und ein neues Skript für das Aktualisieren der Versionsnummer in verschiedenen HTML-Dateien ausgeführt wird (Schuhardt 2023; Billigmann 2023).

Diese Pipeline und das neue Skript wurden prototypisch umgesetzt und sind in den Branches `add-cicd`[13] und `automation`[14] des `renew-web`-Projekts zu finden. Die Umsetzungen wurden auf internen Confluence-Seiten dokumentiert[15][16] und sollten als Ausgangspunkt für zukünftige Automatisierungen genommen werden.

Da die Struktur der `archive.html` seit AOSE22 geändert wurde, müsste die Implementation der automatischen Aktualisierung dieser Datei angepasst werden.

Die JSON-Kontextdatei (s. Abschnitt 5.4.2) wurde in der prototypischen Umsetzung größtenteils hardgecodet (Schuhardt 2023), idealerweise sollten mindestens die Infomationen für die Variablen `versiontype` und `javaversion` der Upstream-Pipeline entnommen werden. Der `versiontype` könnte anhand des jeweiligen Branches (`master` oder `main`) erkannt werden und die `javaversion` aus der Gradle-Datei `build.gradle` gelesen werden.

Bei der prototypischen Umsetzung fehlt die Automatisierung der händischen Anpassungen an den folgenden HTML-Dateien: `history.html`, `issues.html`, `old-issues.html`

[13] https://git.informatik.uni-hamburg.de/tgi/renew-web/-/tree/add-cicd
[14] https://git.informatik.uni-hamburg.de/tgi/renew-web/-/tree/automation
[15] Dokumentation des neuen Skripts: https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?pageId=381222917
[16] Dokumentation der neuen CI/CD-Pipeline: https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?pageId=365789225

und `license.html`. Das Aktualisieren der Jahreszahl in der `license.html` ist hierbei analog zu den durch das neue Skript automatisierten Änderungen in der `renew.hmtl` und der `archive.html` durchzuführen, ebenso das Verschieben der „Known Issues“ aus der Datei `issues.html` in die `old-issues.html`. Für die Dateien `history.html` und `issues.html`, welche das Changelog und die „Known Issues“ beinhalten, muss eine Lösung gefunden werden, woher diese Informationen stammen sollen. Die Changelog-Informationen für die `history.html` könnten der korrespondierenden TEX-Datei des Handbuchs entnommen werden. Die „Known Issues“ jedoch sind an keiner Stelle im `Renew`-GitLabprojekt zu finden, weshalb eine Automatisierung hier z.Zt. nicht möglich ist.

Allgemein sollte geprüft werden, ob die Informationen, die sowohl im Handbuch als auch auf der Webseite hinterlegt werden, für die Webseite automatisiert aus den TEX-Dateien des Handbuchs übernommen werden können. Hierfür müssten die TEX-Dateien in den im `Renew`-GitLabprojekt generierten Artefakten aufgenommen werden, sodass sie für die CI/CD-Pipeline von `renew-web` weiterverarbeitbar sind.

6.5 Weiterhin bestehende Herausforderungen

Es gibt Gegebenheiten und Vorgänge innerhalb des Releaseprozesses von RENEW, die sich nicht ändern lassen bzw. tatsächlich nur manuell gemacht werden können. Diese werden in den folgenden Unterabschnitten beschrieben.

6.5.1 Manuelles Testen und Reviews

Unabhängig vom zukünftigen Ausbau der automatischen Testabdeckung (s. 6.1.1 Testabdeckung), bleiben das explorative Testen des gesamten Systems sowie die Reviews von neuen Codesegmenten und von Releasevorbereitungen weiterhin sehr wichtig. Diese Maßnahmen der Qualitätssicherung sind nicht bzw. nicht ohne Weiteres zu automatisieren (Humble und Farley 2011, S.25).

Durch das Entwerfen und Einhalten von Konventionen und Review-Checklisten (s. Anhänge C und D) können derartige Maßnahmen vereinheitlicht werden, trotzdem bleiben sie fehleranfällig:

- In der Entwicklung von RENEW werden per Konvention alle Änderungen und neuen Funktionalitäten durch Reviewer getestet, bevor sie in den Hauptbranch der Entwicklung gelangen. Eine Anfälligkeit hierbei ist jedoch, dass diese Tests normalerweise in der Form durchgeführt werden, dass die Funktionalität so, wie sie von der ursprünglich entwickelnden Person beschrieben wurde, getestet wird. Diese Art zu Testen bedeutet, dass unvorhergesehene, nicht bedachte Fehler eventuell nicht gefunden werden.
- Die Reviewer sind oft noch nicht lange an der Entwicklung von RENEW beteiligt, weswegen Bugs z.T. erst beim zweiten Schritt des Reviewprozesses gefunden werden, wenn erfahrenere RENEW-Entwickelnde den Code überprüfen, die ein tieferes Verständnis für die Komplexität von RENEW haben. Teilweise werden sie auch erst später durch andere Entwickelnde oder Nutzende gefunden; beispielsweise durch Hauptentwickelnde nachdem das Release bereits gebaut wurde oder sogar erst nach der Veröffentlichung. Auch bei den von dieser Arbeit begleiteten Veröffentlichungen haben sich die Rückmeldungen und Kommentare von Hauptentwickelnden, die sich schon seit Jahren im RENEW-Kontext bewegen, als besonders wertvoll herausgestellt. Diese Personen haben sowohl die benötigten tiefgründigen Fachkenntnisse als auch

den Überblick über das gesamte Softwareprojekt, sodass sie Fehler finden konnten, von denen weniger erfahrene Entwickelnde nicht wussten, dass sie überhaupt auftreten können.

- Die Dokumentation der Konventionen und Checklisten muss stets aktuell gehalten werden. Die Aktualisierung von Dokumentation ist zeitaufwändig und komplex (Humble und Farley 2011, S.6).
- Die Checklisten sind nur effektiv, wenn sie gewissenhaft und genau befolgt werden (Slimmon 2019). Um das Übersehen von Schritten beim Befolgen unwahrscheinlicher zu machen, schlägt Slimmon „do-nothing scripting“ vor: das Entwerfen von Skripten, die eine Person Schritt für Schritt durch eine Checkliste führen, indem sie jeden Schritt auf das Terminal schreiben und von ihr bestätigen lassen, bevor sie fortfahren (Slimmon 2019).

6.5.2 Erstellung der Dokumentation

Die Erstellung der Dokumentation, wie sie zur Zeit für RENEW existiert, lässt sich nicht vollständig automatisieren.

Dazu gehören sowohl das Handbuch und die Readme-Dateien, als auch die dokumentierenden Inhalte auf der Webseite, ausschließlich der bereits erwähnten möglichen automatisierten Erstellung des Changelogs und der automatisierten Aktualisierung von Informationen, die bereits im Code enthalten sind, wie bspw. Versionsnummern (s. 6.2.3 Plugin-bezeichnungen im Handbuch) und gewisse Informationen auf der Webseite (s. 6.4.4 Aktualisierung der Webseiten). Insbesondere die Dokumentation von neuen Funktionalitäten muss weiterhin händisch erstellt werden.

Die Dokumentation der „Known Issues“ auf der Webseite ist ebenfalls nicht bzw. nicht ohne Weiteres automatisierbar. Es wäre zu überlegen, ob eine Möglichkeit existiert, für die „Known Issues“ konsequent Bugtickets in Jira zu erstellen, diese mit einem für die releaserelevanten Bugs festgelegten Label zu versehen und die so gelabelten Tickets automatisiert in die HTML-Datei zu überführen. Hierfür müsste die Beschreibung des Tickets die für die Webseiten benötigten Informationen – eine Beschreibung des Fehlers, eine Auflistung der Umgebungen, in denen er auftritt, den Grund für das Auftreten des Fehlers und, wenn bekannt, eine Übergangslösung für den Fehler – enthalten, damit das Format der „Known Issues“ eingehalten werden kann.

6.5.3 Aktualisierung der Lizenzbedingungen

Die Lizenzbedingungen für RENEW enthalten u.a. die Lizenzbedingungen für von externen Personen entwickelte Teile, welche von RENEW verwendet werden. Die Feststellung davon, ob neue externe Software innerhalb der RENEW-Entwicklung seit dem zuletzt veröffentlichten Release verwendet wird, ist nicht zu automatisieren. Die Informationen darüber müssen von den entsprechenden RENEW-Entwickelnden geliefert und in die Lizenzbedingungen aufgenommen werden. Eine Verschiebung dieses Schrittes aus dem Releaseprozess in den regulären Entwicklungsprozess wurde bereits in Abschnitt 6.2.4 vorgeschlagen.

6.5.4 Flaschenhälse durch Berechtigungen

Gewisse Tätigkeiten sind nur von einzelnen Personen ausführbar, die neben anderen Zuständigkeiten die RENEW-Entwicklung betreuen, wodurch an dieser Stelle ein Flaschenhals besteht.

Bspw. sind hier das Deployment des neuen Release auf dem Server und das manuelle Ausführen des Jobs `release:packages` der CI/CD-Pipeline zu erwähnen. Für den Zugriff auf den Server wird ein Passwort benötigt, welches den meisten RENEW-Entwickelnden nicht bekannt ist; für das manuelle Triggern des Jobs wird die Rolle *Maintainer* innerhalb des `Renew`-GitLabprojekts benötigt, während die meisten RENEW-Entwickelnden die Rolle *Developer* haben.

Die großzügigere Verteilung von Rechten bzw. Rollen würde vermutlich den derzeitigen Releaseprozess zwar beschleunigen, allerdings ginge dies einher mit Sicherheitsrisiken und weniger stark erzwungenen Qualitätskontrollen.

6.5.5 Skillset der Studierenden

Der universitäre Kontext bringt im Bezug auf den Entwicklungs- bzw. Veröffentlichungsprozess von RENEW unterschiedliche Herausforderungen mit sich. Eine dieser Herausforderungen ist, dass die einzelnen Aufgaben an die Kompetenzen der jeweiligen entwickelnden Personen angepasst werden müssen.

Dieser Punkt wird durch die Personen in den agilen Entwicklungsteams der Lehrprojekte berücksichtigt. Es gibt eine Onboarding-Phase, in welcher den Studierenden die grundlegen Kenntnisse und Fähigkeiten für die Entwicklung vermittelt werden (Anonymous Author(s) 2024). Studierende können danach selbst entscheiden, welche Aufgaben sie übernehmen wollen und sich dafür das Wissen selbstständig in den Projektzeiten aneignen; dies gilt auch für Studierende, die sich dafür entscheiden im Releaseteam aktiv zu werden.

Selbst wenn der Releaseprozess weiter automatisiert werden würde, wäre es für eine spätere Anpassung oder Weiterentwicklung der Automatismen nötig, dass die daran arbeitenden Studierenden sich die dafür nötigen Kompetenzen aneignen und die bestehenden Prozesse verstehen. Ein tiefes Verständnis für den gesamten Prozess ist nötig, um die CI/CD-Pipeline und die Automatisierungen zu verbessern (Adams u. a. 2015).

6.5.6 Zeit der Studierenden

Dem universitären Kontext ist eine weitere Herausforderung innerhalb des Entwicklungsprozesses von RENEW geschuldet: die Arbeit an einzelnen Entwicklungen dauert länger als in Teams, die ihre gesamte Arbeitszeit für die Erweiterung, Verbesserung und Pflege ihrer Software aufwenden können.

Diese Gegebenheit ist der Entwicklung im universitären Kontext bzw. Lehrprojekt immanent und unabänderlich, denn Studierenden steht für die Entwicklung innerhalb der Teilnahme an Lehrprojekten nicht ihre gesamte Arbeitszeit zur Verfügung.

Der Umstand, dass die universitäre Softwareentwicklung mehr Zeit benötigt, ist in Überlegungen zu kürzeren Release-Intervallen zu berücksichtigen.

Kapitel 7

Evaluation

In diesem Kapitel wird die vorliegende Arbeit anhand der zuvor festgelegten Anforderungen überprüft. Dabei wird in derselben Reihenfolge vorgegangen, die auch im Anforderungskapitel vorzufinden ist. Es wird erläutert, inwieweit die in den Anforderungen formulierten Ziele erreicht, teilweise erreicht oder übertroffen wurden.

7.1 Durchführung von Releases

Als Teil des jeweiligen Releaseteams der Lehrprojekte wurden sowohl die gemeinsamen Releases von RENEW 2.6 und 4.0 als auch das Release von RENEW 4.1 erfolgreich durchgeführt. Hierbei wurden innerhalb des Releaseteams unterschiedliche Tätigkeiten übernommen, die im Folgenden expliziert werden.

Während des Releaseprozesses aller drei Versionen wurden die folgenden Aufgaben übernommen:

- Rücksprache mit einem Hauptentwickler halten, um die notwendigen Tätigkeiten der Releasevorbereitung zu determinieren, primär vor Beginn der Releasevorbereitungen für 2.6 und 4.0,
- Übernahme von Aufgaben der Rollen PO (s. 2.4.1 Rollen) bzw. Release-Manager, insbesondere die Kommunikation von ausstehenden Aufgaben, Diskussionsbedarfen, Entscheidungen und Reviewanfragen zwischen Releaseteam, Hauptentwicklern, CC-PO und anderen RENEW-Entwickelnden,
- Vorbereitung der Webseiten im GitLab-Projekt `renew-web`, innerhalb derer für die erstmalige Veröffentlichung zweier gleichgestellter MAJOR-Versionen – 2.6 und 4.0 – eine Anpassung des Designs der Startseite entworfen wurde, und
- Review der gesamten Releasevorbereitungen, also der jeweiligen Distribution-Packages und deren Funktionalität, des aktualisierten Handbuchs und der Webseite.

Vor der Veröffentlichung von RENEW 4.0 hat das gesamte Releaseteam zunächst einen releasefähigen Zustand hergestellt. Zu diesem Zweck mussten diverse Plugins an neu festgelegte Konventionen bezüglich des Abhängigkeitsmanagements, der Ordnerstruktur und der Dokumentation überprüft und angepasst werden (Johnsen 2022). Von diesen Anpassungen wurden einige selbst durchgeführt, andere überprüft und bei Bedarf korrigiert. Des Weiteren wurde eine kompliziertere Aufgabe bezüglich des Mappings von Plugin- und Modulnamen innerhalb des *Loader*-Plugins übernommen und gelöst. All diese Tätigkeiten wurden unter Einhaltung der bereits beschriebenen Qualitätssicherungsmaßnahmen

durchgeführt.

Nachdem ein releasefähiger Zustand für die Version 4.0 hergestellt war, wurden in Vorbereitung des Release zusätzlich zu den oben aufgeführten Aufgaben die folgenden im `Renew`-GitLab-Projekt bearbeitet:

- Überprüfung des Umfangs von veröffentlichten Plugins und Distribution-Packages,
- Erhöhung der Plugin-Versionsnummern,
- Überprüfen und Anpassen des Handbuchs,
- Schreiben des Gradle-Tasks für das Distribution-Package Renew IDE,
- Anpassen des Gradle-Release-Tasks bzgl. der generierten Artefakte
- Herstellung von Lauffähigkeit des und Einhaltung der Readme-Konventionen im Mac-OS X Application Bundle

Zusätzlich zu den oben aufgeführten Aufgaben bezüglich aller drei Releases wurden während der Releasevorbereitungen für RENEW 4.1 Anpassungen in den Dateien des Handbuchs vorgenommen.

Darüber hinaus wurde an der Erstellung des veröffentlichten Papers “RENEW: Modularized Architecture and New Features” mitgearbeitet, welches die Neuerungen innerhalb der RENEW-Entwicklung seit der Veröffentlichung der Version 2.5 bis zur Veröffentlichung der Version 4.1 vorstellt (Moldt u. a. 2023a).

Während der aktuell laufenden Releasevorbereitungen von RENEW 4.2 wurde das neue Releaseteam durch die Bereitstellung der Kapitel 4 und 5 sowie die zusätzliche Beantwortung von Fragen unterstützt. Infolgedessen wurde die Dokumentation der Durchführung geringfügig überarbeitet und mit Ergänzungen bzw. Explizierungen versehen.
Zusätzlich wurden Reviews der Releasevorbereitungen für 4.2 durchgeführt. Zunächst wurden die Änderungen in `renew-web` überprüft und kommentiert; später wurde ein umfangreiches Review mittels der neu entworfenen Release Review Checkliste durchgeführt.

Zusammenfassend lässt sich festhalten, dass mehrere RENEW-Releases zusammen mit den Releaseteams erfolgreich durchgeführt wurden, wodurch das Ziel erreicht und durch die Mitautorenschaft an einem Paper und die Unterstützung eines weiteren Releaseprozesses sogar übertroffen wurde.

7.2 Wissensbündelung

Die Bündelung und Strukturierung des Wissens um den Releaseprozess von RENEW ist erfolgt.
Alle benötigten Arbeitsschritte zur Durchführung eines RENEW-Release wurden in Kapitel 5 festgehalten, ergänzend wurden für ein Release wichtige Informationen bzgl. Voraussetzungen und Organisation, Versionierung und Aufbau, Branchingkonventionen und Qualitätssicherung, sowie Buildmanagement und CI/CD-Pipelines in Kapitel 4 festgehalten. Hierfür wurden die eingesetzten Technologien erklärt und es wurde erläutert, welche Akteure die jeweiligen Aufgaben zu übernehmen haben. Zusätzlich wurde eine Übersicht über den Releaseumfang geschaffen, indem die veröffentlichten Plugins der Releases 2.6, 4.0 und 4.1 in Listen aller existierenden RENEW-Plugins der jeweiligen Variante markiert

wurden. Des Weiteren wurde eine Release Review Checkliste entworfen, um das Wissen darum, auf welche Aspekte bei der Überprüfung von Releasevorbereitungen zu achten ist, zu verstetigen.

Auf Basis der Dokumentation wurde das Release von RENEW 4.2 vorbereitet. Hierdurch wurde gezeigt, dass der Anspruch erfüllt werden konnte, den Releaseprozess durch das Explizieren und Strukturieren den Vorgehens für nachfolgende Iterationen desselben von zukünftigen Releaseteams realisierbar zu machen.

7.3 Dokumentation von Herausforderungen

Die bestehenden Herausforderungen innerhalb des Releaseprozesses von RENEW wurden in Kapitel 6 dokumentiert. Hierfür wurden die Erfahrungen, die während der Durchführung der Releases gesammelt wurden, genutzt, Rücksprachen zu offenen Fragestellungen mit dem CCPO der Lehrprojekte gehalten und der Veröffentlichungsprozess durch Literatur gestützt betrachtet.

Die offenen Fragestellungen wurden gegliedert in die Themenbereiche Technische Schulden, Umsetzung, Organisation und Automatisierung. Ausgehend von der Dokumentation dieser Herausforderungen konnten Vorschläge für den Umgang mit ihnen erarbeitet werden.

Zusätzlich wurden weiterhin bestehende Herausforderungen dokumentiert, welche auch nach möglichen Anpassungen bestehen bleiben werden. Die meisten dieser Umstände basieren auf der Entwicklung innerhalb des universitären Kontextes. Manuelles Testen und Reviews sind unabhängig vom universitären Kontext ratsam, wenn eine komplexe Software von einem relativ großen Team und ohne ausreichende automatisierte Testabdeckung entwickelt wird. Die Erstellung der Dokumentation, wie sie zur Zeit für RENEW existiert, ist unabhängig vom Kontext eine Aufgabe, die nicht vollautomatisiert werden kann und aufwändig bleibt.

Darüber hinaus sollte die Dokumentation der Herausforderungen – insbesondere innerhalb der Umsetzung – zukünftige Releaseteams darauf aufmerksam machen, welche Fallstricke in der Durchführung von RENEW-Releases bestehen.

Releaseprozesse sind individuell, kompliziert und finden in einer komplexen Umgebung statt. Aus diesem Grund lässt sich nur eingeschränkt beurteilen, ob die Sammlung und Dokumentation der offenen Fragestellungen vollständig ist.

Da die Herausforderungen anhand gemachter Erfahrungen und im Vergleich zu von der Literatur betrachteten Praktiken gesammelt wurden, ist davon auszugehen, dass wenigstens diejenigen, die den Prozess am erheblichsten beeinflusst haben bzw. am stärksten von Best Practices aus der Literatur abweichen, Betrachtung erhalten haben, und somit das Ziel mindestens teilweise erfüllt wurde.

7.4 Betrachtung möglicher Verbesserungen

Die Betrachtung möglicher Anpassungen innerhalb des Releaseprozesses erfolgte ebenfalls in Kapitel 6.

Die Vorschläge für die offenen Fragestellungen wurden basierend auf Best Practices in Literatur und Praxis gemacht. Gleichzeitig wurde darauf geachtet, den realen Kontext von

RENEWs Entwicklung und Veröffentlichung zu berücksichtigen, sodass die vorgeschlagenen Anpassungen tendenziell konservativ sind. Es wurde jeweils begründet, warum bzw. wie der jeweilige Vorschlag die identifizierte Herausforderung adressiert.

Die Anforderung konnte also erfüllt werden, allerdings wird sich erst bei einer künftigen Umsetzung zeigen, wie effektiv die Vorschläge den Releaseprozess unterstützen können.

Einige Verbesserungsvorschläge wurden bereits gänzlich bzw. zum Teil umgesetzt.

Die innerhalb dieser Arbeit geschaffene Dokumentation der Releasedurchführung und die Release Review Checkliste stellen bereits die Umsetzung von möglichen Verbesserungen dar. Die Dokumentation entspricht einer Release Checkliste, die für die Vorbereitung künftiger Releases genutzt werden kann. Die Release Review Checkliste kann die Vorbereitungen ergänzend unterstützen. Durch die Verwendung beider Checklisten sollte die Releasedurchführung an Zuverlässigkeit gewinnen. Allerdings sind Dokumentationen aufwändig zu pflegen, da sie schnell an Aktualität verlieren oder unvollständig sein können (Humble und Farley 2011, S.6). Daher sollte die Dokumentation nach der Automatisierung der Prozesse obsolet gemacht werden. Dennoch ist die Dokumentation eine reale Verbesserung des Releaseprozesses und ein sinnvoller erster Schritt hin zur Automatisierung (Barnett 2020).

Für die Automatisierung der Erstellung der Webseiten wurde bereits eine prototypische Umsetzung innerhalb des Releaseteams von AOSE22 geschaffen.

Die Umsetzungen anderer Vorschläge sind bereits Thema geplanter Abschlussarbeiten bzw. Vorhaben innerhalb der Lehrmodule, in denen RENEW entwickelt wird. Dies betrifft die Umstrukturierung der Dokumentation, die automatisierte Erstellung von Changelogs, die Veröffentlichung von Nightly Builds und den Einsatz von Sonargraph im Releaseprozess.

7.5 Gesamtevaluation

Allgemein sollten die erfolgreiche Durchführung, eine detaillierte Dokumentation und die kritische Betrachtung des Releaseprozesses einer komplexen, im universitären Kontext entwickelten Software geleistet werden. Diese Ziele konnten erreicht werden.

Es ist festzuhalten: Releaseprozesse haben sich als sehr individuell erwiesen, dennoch kann diese Arbeit ein Anwendungsbeispiel sein, wie ein Releaseprozess einer komplexen OSS innerhalb des universitären Kontextes gestaltet und erfolgreich umgesetzt werden kann, und es konnten Herausforderungen innerhalb und Möglichkeiten zur Verbesserung eines solch komplexen Systems aufgezeigt werden.

Kapitel 8

Schluss

Dieses Kapitel liefert zunächst eine Zusammenfassung der vorliegenden Arbeit. Abschließend wirft es einen Ausblick auf mögliche zukünftige, auf diese Arbeit aufbauende Themen und Weiterentwicklungen.

8.1 Zusammenfassung

In Kapitel 1 werden die Motivation für die Erstellung dieser Arbeit, die bearbeitete Fragestellung und Zielsetzung sowie der Aufbau der Arbeit präsentiert.

Kapitel 2 beschreibt für diese Arbeit wichtige Grundlagen. Hierfür werden die Software RENEW, der universitäre Kontext, der Open-Source-Begriff, das Projektmanagement-Framework Scrum, Grundbegriffe der Versionsverwaltung mit Git, der Begriff Releaseprozess, die Konzepte CI und CD sowie die Anwendung GitLab vorgestellt.

Kapitel 3 führt die Anforderungen an die vorliegende Arbeit auf. Diese betreffen die Durchführung von Releases, die Wissensbündelung bzgl. des Releaseprozesses, die Dokumentation der Herausforderungen und die Betrachtung möglicher Verbesserungen. Zusätzlich werden die Grenzen dieser Arbeit erläutert.

Mit Kapitel 4 wird der erste Teil der Wissensbündelung bzgl. des Releaseprozesses von RENEW geliefert. Diesbezüglich werden die Voraussetzungen und die Organisation eines Release, die Produktlinien, die Versionierung, der Aufbau der veröffentlichten Artefakte und die Lizenzierung von RENEWs Releases, die Branching-Konventionen, die Qualitätssicherungsmaßnahmen bei Änderungen auf dem Hauptbranch, das Buildmanagement und der Einsatz von CI/CD-Pipelines innerhalb der Entwicklung von RENEW beschrieben.

Darauffolgend wird in Kapitel 5 der zweite Teil der Wissensbündelung bzgl. des Releaseprozesses von RENEW geleistet, indem alle Schritte, die für die Durchführung des Releaseprozesses nach der Herstellung eines releasefähigen Zustands befolgt werden müssen, aufgelistet werden. Diese umfassen die Erstellung des Changelogs, die Überprüfung und Erhöhung der Versionsnummern, die Aktualisierung der RENEW-Dokumentation, die Vorbereitung der Webseiten und das Deployment des Release.

In Kapitel 6 werden offene Fragestellungen und mögliche Anpassungen innerhalb des Releaseprozess untersucht. Das Kapitel ist dabei in die Themenbereiche technische Schulden, Umsetzung, Organisation, Automatisierung und weiterhin bestehende Herausforderungen gegliedert. Innerhalb der ersten vier Themenbereiche werden für jede beleuchtete Fragestellung zunächst die Problembeschreibung und darauffolgend Vorschläge für den Umgang mit der Herausforderung präsentiert.

Kapitel 7 diskutiert die Erreichung der in Kapitel 3 festgelegten Ziele und Anforderungen an diese Arbeit.

Das die Arbeit abschließende Kapitel 8 liefert neben dieser Zusammenfassung einen Ausblick auf mögliche zukünftige, auf diese Arbeit aufbauende Themen und Weiterentwicklungen.

8.2 Ausblick

Der Fokus dieser Arbeit lag auf der Durchführung, Dokumentation und Betrachtung des Releaseprozesses einer komplexen, im universitären Kontext entwickelten Software. Aus den in Kapitel 6 festgehaltenen Überlegungen zu offenen Fragestellungen im Rahmen der Veröffentlichung ergeben sich unmittelbar Themen für zukünftige Arbeiten und Weiterentwicklungen.

Die Ausweitung der Testabdeckung für RENEWs Code sollte vorangetrieben werden, denn automatisierte Tests bilden die Grundlage für die Erhöhung des Automatisierungsgrades des Releaseprozesses. Der Prozess der Testerstellung wurde bereits in der Arbeit "Qualitätssicherung der Software Renew durch automatisierte Tests und Testumgebung" begonnen, wird aber noch viel Zeit in Anspruch nehmen, da er mit hohem Aufwand verbunden ist. Dabei sind Abwägungen bzgl. des Zeitaufwands und des zu erzielenden Nutzens zu treffen, da einige Plugins bzw. Codesegmente seit vielen Jahren keine Änderungen erfahren haben oder aber der jeweilige Code nicht als zentraler Bestandteil von RENEW betrachtet werden kann, da es sich um Prototypen handelt. Wenigstens für die veröffentlichten Teile von RENEW ist eine vollständige Testabdeckung jedoch höchst erstrebenswert.

Um den Releaseprozess quantifizierbar zu machen, müssen Metriken eingeführt werden, an denen dieser gemessen werden kann. Nach erfolgreicher Implementation von derartigen Messungen, wäre es sinnvoll Grenzwerte einzuführen, anhand derer entschieden werden kann, ob ein Releasekandidat für die Veröffentlichung bereit ist oder nicht. Dabei ist abzuwägen, welche Metriken zu diesem Zweck zu erheben sind. Die Ausweitung des Einsatzes von Sonargraph wird innerhalb der RENEW-Entwicklung bereits geplant.

Innerhalb dieser Arbeit wurde erläutert, dass RENEWs Dokumentation schlechte Wartbarkeit, Redundanzen und Obsoleszenzen aufweist. Die Konzeption einer redundanzfreien Dokumentation und der Entwurf eines aktuellen Architecture-Guides stellen interessante Aufgabenstellungen dar.

Die Organisation eines Releaseprozesses innerhalb eines Lehrprojekts, in dem zeitgleich viele weitere Entwicklungen stattfinden, hat sich als sehr herausfordernd herausgestellt. Mögliche organisatorische Unterstützungen des Releaseprozesses sollten in Zukunft untersucht und ihre Umsetzung evaluiert werden. Diese könnten die bereits erwähnten Branching-Konventionen bzgl. eines neuen Release-Branches und Codefreeze-Zeiten beinhalten.

Innerhalb von RENEW 4.x-Versionen wurden modularisierte Plugins der klassischen 2.x-Versionen veröffentlicht. Die Modularisierung von RENEW ist jedoch noch nicht abgeschlossen. Langfristig sollten alle bestehenden RENEW-Plugins nach Daschkewitsch 2019, Hansson 2020, Janneck 2021 und Johnsen 2022 modularisiert werden, sodass die klassische Version komplett durch die modularisierte abgelöst werden kann. Veröffentlichte Plugins der klassischen Version sollten hierbei priorisiert werden und der Nutzen der Modularisierung von nicht veröffentlichten, prototypischen Plugins abgewogen werden.

Bugs werden i.d.R. auch noch nach einem Release entdeckt. RENEW wird zur Zeit mit einer statischen „Known Issues"-Webseite veröffentlicht. Es wäre interessant zu untersuchen, inwiefern die Kommunikation von bekannten Softwarefehlern an die Nutzenden dynamisch erfolgen kann, sodass auch später entdeckte Fehler sofort an die Nutzenden kom-

muniziert werden können.

Innerhalb des Releaseprozesses von RENEW bestehen noch diverse Automatisierungsmöglichkeiten. Hierbei ist die automatisierte Erstellung von Changelogs, die Vollautomatisierung der Erstellung bzw. Aktualisierung der Webseite und die Umsetzung einer automatisierten CD-Pipeline zu erwähnen. Die Erstellung von Nightly Builds könnte nachfolgend auf all diese Automatisierungen aufbauen.

In der Arbeit Heinze 2022 wurde die Aufteilung von RENEW auf verschiedene Repositories beschrieben. Das damit entstehende RENEW 5.0 wurde bisher nur prototypisch umgesetzt. Die vollständige Umsetzung wird Verwaltungs- und Organisationsaufwand erzeugen, jedoch wird der Releaseprozess nach der Aufteilung voraussichtlich übersichtlicher werden, da Releases für einzelne Plugins durchgeführt werden könnten, anstatt für RENEW insgesamt. Eine Voraussetzung für diese Umsetzung ist, dass Refaktorisierungen über mehrere Repositories hinweg erfolgen können. In zukünftigen Arbeiten kann der Releaseprozess für das aufgeteilte RENEW angepasst und der angepasste Prozess untersucht werden.

Anhang A

Open-Source-Definition

Die zehn Kriterien, die in den Distributionsbedingungen von OSS gegeben sein müssen, damit sie als Open Source laut der OSI gilt, lauten wie folgt (aus dem Englischen übersetzt mit Unterstützung von DeepL[1]) (The Open Source Initiative 2023b):

1. **Freie Weiterverbreitung**

   Die Lizenz darf niemanden daran hindern, die [ursprüngliche] Software als Bestandteil einer aggregierten Softwaredistribution, die Programme aus verschiedenen Quellen enthält, zu verkaufen oder weiterzugeben. Die Lizenz darf für derartige Verkäufe keine Lizenzgebühren oder sonstige Entgelte verlangen.

2. **Quellcode**

   Das Programm muss den Quellcode enthalten und sowohl die Distribution des Quellcodes als auch der kompilierten Form erlauben. Wenn ein Produkt in irgendeiner Form nicht mit dem Quellcode vertrieben wird, muss es eine gut publizierte Möglichkeit geben, den Quellcode für nicht mehr als angemessene Reproduktionskosten zu erhalten, vorzugsweise durch kostenloses Herunterladen über das Internet. Der Quellcode muss in der bevorzugten Weise vorliegen, in welcher ein:e Programmier:in das Programm ändern würde. Vorsätzlich verschleierter Quellcode ist nicht zulässig. Zwischenformen wie die Ausgabe eines Vorverarbeitungsprozessors oder eines Konvertierungsprogramms sind nicht erlaubt.

3. **Abgeleitete Werke**

   Die Lizenz muss Modifikationen und abgeleitete Arbeiten zulassen und deren Distribution unter denselben Bedingungen erlauben wie die Lizenz der Originalsoftware.

4. **Integrität des Quellcodes des Urhebers/der Urheberin**

   Die Lizenz darf die Distribution von Quellcode in abgewandelter Form nur dann einschränken, wenn die Lizenz die Weitergabe von „Patch-Dateien“ zusammen mit dem Quellcode erlaubt, um das Programm zur Bauzeit zu verändern. Die Lizenz muss explizit die Distribution von Software erlauben, die aus modifiziertem Quellcode erstellt wurde. Die Lizenz kann vorschreiben, dass abgeleitete Werke einen anderen Namen oder eine andere Versionsnummer als die originäre Software tragen müssen.

5. **Keine Diskriminierung gegen Personen oder Gruppen**

   Die Lizenz darf gegen keine Person oder Personengruppe diskriminieren.

[1] https://www.deepl.com/translator

6. **Keine Diskriminierung gegen Einsatzbereiche**

   Die Lizenz darf niemanden darin einschränken, das Programm in einem bestimmten Tätigkeitsbereich einzusetzen. Z.B. darf sie die Verwendung des Programms in einem Unternehmen oder in der Genforschung nicht ausschließen.

7. **Verbreitung der Lizenz**

   Die mit dem Programm verbundenen Rechte müssen für alle, an die das Programm weiterverteilt wird, gelten, ohne dass von diesen Parteien die Ausführung einer zusätzlichen Lizenz gefordert wird.

8. **Die Lizenz darf nicht produktspezifisch sein**

   Die mit dem Programm verbundenen Rechte dürfen nicht davon abhängig gemacht werden, dass das Programm Teil einer bestimmten Softwaredistribution ist. Falls das Programm aus dieser Distribution extrahiert und im Rahmen der Lizenzbedingungen verwendet oder weitergegeben werden sollte, sollten alle Parteien, an die das Programm weitergegeben wird, dieselben Rechte haben, die in Verbindung mit der ursprünglichen Softwaredistribution gewährt werden.

9. **Die Lizenz darf andere Software nicht einschränken**

   Die Lizenz darf keine Einschränkungen für andere Software enthalten, die zusammen mit der lizenzierten Software vertrieben wird. Bspw. darf die Lizenz nicht darauf bestehen, dass alle anderen Programme, die über dasselbe Medium verteilt werden, OSS sein müssen.

10. **Die Lizenz muss technologieneutral sein**

    Keine Bestimmung der Lizenz darf sich auf eine bestimmte Technologie oder eine bestimmte Art der Schnittstelle beziehen.

Anhang B

Übersicht zu veröffentlichten Plugins

Die Anhänge B.1 und B.2 geben einen Überblick darüber, welche RENEW-Plugins in den unterschiedlichen Produktlinien – 2.x bzw. klassisch und 4.x bzw. modular – existieren und welche davon wiederum in den unterschiedlichen Distribution-Packages, als optionale Plugins und/oder als Quelltext (Source) in den letzten drei Releases veröffentlicht wurden.

B.1 In RENEW 2.6 veröffentlichte Plugins

Zu der Veröffentlichung von RENEW 2.6 gehören die Distribution-Packages Base, Analysis, CCPN, IDE und Plugin Development. Außerdem gehören auch die optionalen Plugins und der veröffentlichte Quellcode dazu. Die zusätzlichen optionalen Plugins („Extra optional Plugins“) sind offiziell nicht veröffentlicht.

Die Abbildung B.1 gibt einen Überblick darüber, welche Plugins in welchen dieser Bestandteile des von RENEW 2.6 enthalten sind. In der obersten Zeile sind die Distributions-Packages etc. aufgelistet, in der ersten Spalte alle 77 Plugins, die auf dem `master`-Branch von RENEW aktuell[1] vorhanden sind.

[1] Commit-SHA: cdbe6304120d2e288558942a4138c23b6dd6eca8

	Base	Analysis	CCPN	IDE	Plugin Development	Optional Plugins	Extra optional Plugins	Source
Access							2.6	
Android								
AppleUI						2.6		2.6
Averto								
Batch								
BPMN								
BPMNRoundtrip								
Catch								
CCPN			2.6					2.6
CH	2.6	2.6	2.6	2.6	2.6			2.6
CNFormalism								
Composition								
Console	2.6	2.6	2.6	2.6	2.6			2.6
Diagram							2.6	
Distribute								
Eclipse								
ElementaryObjectNetSystems								
Export	2.6	2.6	2.6	2.6	2.6			2.6
Extras								
FA				2.6		2.6		2.6
FAFormalism				2.6		2.6		2.6
Formalism	2.6	2.6	2.6	2.6	2.6			2.6
FormalismGui	2.6	2.6	2.6	2.6	2.6			2.6
FS		2.6		2.6		2.6		2.6
Gui	2.6	2.6	2.6	2.6	2.6			2.6
GuiPrompt						2.6		2.6
HierarchicalWorkflowNets								
ICWorkflow								
ImageNets								
ImageNetDiff		2.6		2.6		2.6		2.6
JGitLibs				2.6		2.6		2.6
Loader	2.6	2.6	2.6	2.6	2.6			2.6
Logging	2.6	2.6	2.6	2.6	2.6			2.6
Lola		2.6				2.6		2.6
Lola2						2.6		2.6
MariaNets								
MiniMap								
Misc	2.6	2.6	2.6	2.6	2.6			2.6
Model								
ModelChecker								
ModelConverter								
MongoDB								
MVS								
Navigator	2.6	2.6	2.6	2.6	2.6			2.6
NavigatorDiff		2.6		2.6				2.6
NavigatorGit				2.6				2.6
NavigatorSVN								
NavigatorVC				2.6				2.6
NetAnalysis							2.6	
NetComponents	2.6	2.6	2.6	2.6	2.6			2.6
NetDoc								
NetTest								
Ontology								
PluginDevelopment		2.6		2.6	2.6			2.6
PnConverter								
Prompt								
Refactoring		2.6		2.6		2.6		2.6
RefactoringOntology								
Remote	2.6	2.6	2.6	2.6	2.6			2.6
RenewAnt		2.6		2.6	2.6			2.6
RenewGrass								
RNRG			2.6			2.6		2.6
Scheme								
SDNet								
Simulator	2.6	2.6	2.6	2.6	2.6			2.6
SL								
SonarEditor								
SonarNetPrototype								
Splashscreen	2.6	2.6	2.6	2.6	2.6			2.6
SvnLibs								
Tablet						2.6		2.6
Test								
Tutorial								
UnitEdComponents								
Util	2.6	2.6	2.6	2.6	2.6			2.6
Velocity		2.6		2.6	2.6			2.6
WFNet							2.6	
Workflow							2.6	
XRN						2.6		2.6

Tabelle B.1: In RENEW 2.6 veröffentlichte Plugins

B.2 In RENEW 4.0 und 4.1 veröffentlichte Plugins

Zu der Veröffentlichung von RENEW 4.0 und 4.1 gehören die Distribution-Packages Base und IDE. Außerdem gehören auch die optionalen Plugins und der veröffentlichte Quellcode dazu. Die zusätzlichen optionalen Plugins („Extra optional Plugins“) sind offiziell nicht veröffentlicht.

Die Abbildung B.2 gibt einen Überblick darüber, welche Plugins in welchen dieser Bestandteile des von RENEW 4.0 bzw. 4.1 enthalten sind. In der obersten Zeile sind die Distributions-Packages etc. aufgelistet, in der ersten Spalte alle 65 Plugins, die auf dem `main`-Branch von RENEW aktuell[2] vorhanden sind und modularisiert sind.

[2] Commit-SHA: fb0df3ad37d0367b4268c8f9e9dd374cc3ed9432

	Base	IDE	Optional Plugins	Extra optional Plugins	Source
Access				4.0 und 4.1	
AppleUI			4.0 und 4.1		4.0 und 4.1
Averto					
Batch					
BPMN					
Catch					
CCPN					4.0 und 4.1
CH	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
CNFormalism					
Composition					
Console	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
Diagram				4.0 und 4.1	4.0 und 4.1
Distribute					
Export	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
Extras					4.0 und 4.1
FA		4.0 und 4.1	4.0 und 4.1		4.0 und 4.1
FAFormalism		4.0 und 4.1	4.0 und 4.1		4.0 und 4.1
Formalism	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
FormalismGui	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
FS		4.0 und 4.1	4.0 und 4.1		4.0 und 4.1
Gui	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
GuiPrompt			4.0 und 4.1		4.0 und 4.1
HierarchicalWorkflowNets					
ImageNetDiff		4.0 und 4.1	4.0 und 4.1		4.0 und 4.1
InterfaceNets					
Loader	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
Logging	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
MailNotificationNetComponents					
MiniMap		4.0 und 4.1	4.1		4.0 und 4.1
Misc	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
Momoc					
MomocGui					
MVS					
Navigator	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
NavigatorDiff		4.0 und 4.1	4.0 und 4.1		4.0 und 4.1
NavigatorGit		4.0 und 4.1	4.0 und 4.1		4.0 und 4.1
NavigatorSVN					
NavigatorVC		4.0 und 4.1	4.0 und 4.1		4.0 und 4.1
NetAnalysis				4.0 und 4.1	
Netcomponents	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
NetSigner					
OccourrenceNet					
Ontology					4.0 und 4.1
PhoneShop					
PrimeCompute					
PTChannel	4.1	4.1			4.1
Refactoring			4.0 und 4.1		4.0 und 4.1
Remote	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
RenewAnt					4.0 und 4.1
RGBase					4.0 und 4.1
RNRG					
Simulator	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
SiphonTrapChecker					
SL					
Splashscreen	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
StatusCollector					
Suite					
SuiteCreator					
SuiteGui					
SvnLibs					
Tablet			4.0 und 4.1		4.0 und 4.1
Util	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
WFNet				4.0 und 4.1	
WindowManagement	4.0 und 4.1	4.0 und 4.1			4.0 und 4.1
Workflow				4.0 und 4.1	

Tabelle B.2: In RENEW 4.0 und 4.1 veröffentlichte Plugins

Anhang C

Paose: Code und Thesis Checkliste (Clasen und Hansson 2023)

Die Checkliste wurde von Clasen und Hansson entworfen und in LaTeX erstellt. Für die Checkliste existiert ein GitLab-Projekt unter `https://git.informatik.uni-hamburg.de/tgi/theses/paose-code-and-thesis-checkliste`. Die hier wiedergegebene Version ist datiert auf den 08.12.2023[1].

C.1 Vorgehensweise

Die Reviews dienen der Verbesserung der Qualität. Deshalb wird grundlegend immer gefordert, dass alle Beschreibungen und Texte aussagekräftig sein müssen.

Dadurch, dass RENEW *als Open-Source veröffentlicht wird muss sämtliche Entwicklung, die mit* RENEW *und seinem Ökosystem zu tun hat, auf* ***Englisch*** *verfasst werden.* Hiervon sind insbesondere Commit-Messages, Tickets, Merge Requests und die Dokumentation betroffen. Grundregel hierbei ist es, dass alles was potentiell veröffentlicht wird auf Englisch verfasst werden muss.

Für ein Review müssen folgende Schritte durchgeführt werden:

0. Trage dich sowohl im Ticket als auch im Merge Request als Reviewer ein.
1. Sind die Merge Request Konventionen C.2 eingehalten worden?
2. Sind die Jira Ticket-Konventionen C.3 eingehalten worden?
3. Sind alle Kriterien der Funktionsprüfung C.4 erfüllt?
4. Sind die Git-Konventionen C.6 eingehalten worden?
5. Sind die Java-Konventionen C.5 eingehalten worden?
6. Sind die zusätzlichen Code-Konventionen C.7 eingehalten worden?
7. Sind die Konventionen für die Dokumentation C.8 eingehalten worden?

Sollte einer der angegebenen Punkte nicht erfüllt sein, wird die Person, die den Merge Request erstellt hat, informiert, was noch nachgepflegt werden muss. Nachdem die angegebenen Punkte nachgepflegt worden sind, müssen die obigen Punkte erneut überprüft werden, dieses Vorgehen wird solange wiederholt, bis die Anforderungen erfüllt sind.

[1] Etwaige Fehler in Grammatik oder Rechtschreibung existieren exakt so in dem hier zitierten Original und werden nicht korrigiert.

C.2 Merge Request - Konventionen

Die Konventionen für Merge Requests befinden sich unter `https://tgipm.informatik.uni-hamburg.de/confluence/display/RENEWDEV/Merge+Request`. Für das Code-Review sind hierbei folgende Punkte zu überprüfen, die letzten beiden Punkte sind hierbei optional:

- ☐ Ist das zugehörige Ticket im Merge Request enthalten?

Optional Enthält der Merge Request Informationen über die neue Funktionalität?

Optional Enthält der Merge Request Informationen darüber wie die neue Funktionalität zu testen ist?

C.3 Jira Ticket - Konventionen

Die Konventionen für die Jira sind unter `https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?pageId=99581982` zu finden. Für das Code-Review muss Folgendes überprüft werden:

- ☐ Wurde innerhalb des Haupttickets beschrieben, was gemacht worden ist?
- ☐ Enthält das Hauptticket ein Label des Formats „<projekt-name>::<team-name>“?
- ☐ Enthält das Hauptticket ein Label des Formats „<projekt-name>::<domain-name>“?
- ☐ Wurde innerhalb jedes Untertickets beschrieben, wozu was gemacht worden ist?
- ☐ Enthält jedes Unterticket ein Label des Formats „<projekt-name>::<team-name>“?
- ☐ Enthält jedes Unterticket ein Label des Formats „<projekt-name>::<domain-name>“?
- ☐ Wurde innerhalb des Haupttickets beschrieben, wie der Merge Request getestet werden kann?
- ☐ Wurde das Hauptticket bis auf das Review sauber abgeschlossen?

Optional Ist der zugehörige Merge Request im Ticket enthalten?

C.4 Funktionsprüfung

Für die Funktionsprüfung sind folgende Punkte zu überprüfen:

- ☐ Das Bauen der Funktionalität enthält keine Errors.
- ☐ Das Bauen der Funktionalität enthält keine Warnings.
- ☐ Sind die Tests der Funktionalität nach der Dokumentation im Jira Ticket erfolgreich?
- ☐ Gibt es Beispiele, in denen Renew die neuen Features auch benutzt?
- ☐ Funktionieren alle existierenden Beispiele der Funktionalität?
- ☐ Sind die Code-Änderungen mit Unit-Tests getestet worden?

- ☐ Wenn mehrere Klassen/Module beteiligt sind, wurden die entsprechenden Änderungen mit Integrationstests getestet?
- ☐ Funktionieren alle Unit-Tests?
- ☐ Funktionieren alle Integrations-Tests?
- ☐ Funktionieren alle Gray Box-Tests?

C.5 Java - Konventionen

Die Konventionen für die Programmiersprache Java in Renew befindet sich unter `https://tgipm.informatik.uni-hamburg.de/confluence/display/RENEWDEV/Java`. Für das Code-Review sind hierbei die folgenden Punkte zu überprüfen:

- ☐ Sind die Javadoc-Tags wie gefordert vorhanden?
- ☐ Wurden die Namenskonventionen eingehalten?
- ☐ Wurde @link statt @see als Javadoc-Tag verwendet?
- ☐ Kann die Javadoc ohne Errors gebaut werden? (./gradlew Gui:javadoc)
- ☐ Entspricht die Ordnerstruktur den Konventionen?
- ☐ Ist der Code verständlich geschrieben?
- ☐ Ist der Code leserlich geschrieben?
- ☐ Sind die entsprechenden Kommentare in der Modul-Info enthalten?
- ☐ Sind nur notwendige Imports benutzt? (so wenig wie möglich)

C.6 Git - Konventionen

Die Konventionen für das Git in Renew befinden sich unter `https://tgipm.informatik.uni-hamburg.de/confluence/display/RENEWDEV/Git`. Für das Code-Review sind hierbei die folgenden Punkte zu überprüfen:

- ☐ Sind alle Personen, welche Commits erstellt haben, auch Autor:innen dieser Commits?
- ☐ Entsprechen die Commit-Nachrichten den Konventionen? (Überprüfe dies lediglich bei offenen Merge-Requests)
- ☐ Entspricht der Name des Branches den Konventionen?
- ☐ Sind nur nützliche Tags hinzugefügt worden?

C.7 Zusätzliche Code - Konventionen

Zusätzliche Code Konventionen in Renew sind unter https://tgipm.informatik.uni-hamburg.de/confluence/display/RENEWDEV/Java zu finden. Für das Code-Review sind hierbei die folgenden Punkte zu überprüfen:

- ☐ Überprüfe die module-info, plugin.cfg und die build.gradle daraufhin, dass die jeweils eingetragenen Abhängigkeiten übereinstimmen. Sollte dies nicht der Fall sein, muss analysiert werden, ob dies beabsichtigt war oder nicht. Falls dies nicht beabsichtigt war, müssen die eingetragenen Abhängigkeiten angepasst werden.
- ☐ Besitzt der Plugin-Name des Plugins den Präfix „Renew Plugin-Name “
- ☐ Werden die build.xml Dateien für das Bauen der Software noch benötigt?

C.8 Dokumentation - Konventionen

Die Konventionen für die READMEs in Renew lassen sich am einfachsten anhand einer Beispiel README erklären, diese ist unter https://git.informatik.uni-hamburg.de/tgi/Renew/-/blob/main/sample_readme.md zufinden. Hierfür muss im Code-Review folgendes überprüft werden:

- Anpassung eines Plugins
  - ☐ Wurde die README angepasst, falls anwendbar?
  - ☐ Wurde der Handbucheintrag angepasst, falls anwendbar?
  - ☐ Wurde die Dokumentation des Plugins in Confluence aktualisiert, falls anwendbar?
- Erstellung eines neuen Plugins
  - ☐ Entspricht der Aufbau der README des Plugins dem der Beispiel README?
  - ☐ Wurde ein Handbucheintrag erstellt?
  - ☐ Wurde das Plugin in Confluence dokumentiert?

Anhang D

Release Review Checkliste

Die Qualitätssicherung beim Releaseprozess wurde mit einem Hauptentwickler besprochen. Hierfür wurde die Frage gestellt, auf welche Aspekte er bei dem Review der Releasevorbereitungen – also des für die Veröffentlichung vorbereiteten Release selbst, zu dem der Code, die Builds, die Readme-Dateien und das Handbuch gehören, und der aktualisierten Webseite – achtet. Basierend auf seiner Antwort wurden die folgende Checkliste zusammengestellt. Zusätzlich wurden Anmerkungen des CCPO bzw. Modulverantwortlichen der Lehrprojekte eingepflegt.

D.1 Funktionalitäten

- ☐ Wurden alle angefassten Funktionalitäten kritisch getestet?
- ☐ Wurden alle angefassten Funktionalitäten daraufhin betrachtet, ob sie an anderer Stelle Probleme verursachen?[1]

Optional Wurden alle im Handbuch beschriebenen Funktionalitäten getestet?[2]

D.2 Systeme

- ☐ Wurde beim Testen der Funktionalitäten darauf geachtet, sie auf verschiedenen Systemen zu testen?
- ☐ Wurden Funktionalitäten, die Pfadkonfigurationen berücksichtigen (z.B. im Navigator), auf Windows und Linux/MacOS (unixartig) getestet?
- ☐ Wurden das MacOS Application Bundle und das DMG getestet? Zu prüfen (mindestens):
  - Sieht das Hintergrundbild normal aus?
  - Lässt sich die .app öffnen?
  - Funktioniert die Desktopintegration? Dafür:
    * Netz malen und nicht speichern
    * RENEW mit `Cmd+Q` beenden

[1] Hierfür muss bekannt sein, wie die Codesegmente miteinander interagieren.

[2] In der Regel ist dieser Schritt für einen MINOR- oder PATCH-Release zu aufwändig, ist aber bei einem MAJOR-Release durchaus sinnvoll.

    * Wenn abgefragt wird, ob das Netz gespeichert werden soll, funktioniert sie
    * Wenn RENEW einfach beendet wird, funktioniert sie nicht
  - Ist GuiPrompt unter den Plugins aufgeführt?
    * Wenn ja, dann ist es laut einem Hauptentwickler im Menü unter Plugins → GuiPrompt zu finden
    * Bei RENEW 4.0 und 4.1 ist es allerdings nur in den optionalen Plugins enthalten

☐ Wurden die Installation und Ausführung der Binärversionen auf Windows, Linux und MacOS getestet? Dafür jeweils[3]:

  - Distribution-Package entpacken
  - `installrenew` ausführen
  - `renew` ausführen
  - `loadrenew` ausführen

☐ Wurde unter Windows die Verknüpfung mit .rnw-Dateien getestet? Dafür:

  - Registry-Einträge mit `addregistry.reg` hinzufügen[4]
  - Danach eine .rnw-Datei durch Doppelklicken öffnen
  - Wenn RENEW geöffnet und das Netz angezeigt wird, funktioniert die Verknüpfung

☐ Wurde unter Mac und Windows getestet, ob RENEW-Netze die richtigen Icons haben?

☐ Wurde unter Mac und Windows getestet, ob Netze in eine bestehende RENEW-Instanz geladen werden?

  - Wenn für jedes Netz eine neue Instanz geöffnet wird, ist das nicht das gewünschte Verhalten

D.3 Versionsnummern (Renew, Plugins und Java)

☐ Wurde überall die neue RENEW-Versionsnummer eingetragen? Zu prüfen (mindestens):

  - Steht die korrekte Versionsnummer im About-Dialog, welcher in RENEW unter dem Help-Menüpunkt aufgerufen werden kann?
  - Steht die korrekte Versionsnummer in der Build-Datei `release-tasks.gradle`?
  - Steht die korrekte Versionsnummer an den ensprechenden Stellen im Handbuch, in den Readme-Dateien und in den Dateien der Webseite?[5]

Tip `grep` nutzen, um den gesamten Quellcode nach der alten Versionsnummer zu durchsuchen. Alternativ: „Find in files...“ von IntelliJ nutzen.

[3] Ausführlichere Anweisungen sind im Handbuch unter „Platform-specific Hints“ zu finden.

[4] Ausführliche Anweisungen sind im Handbuch unter „Windows“ unter „Platform-specific Hints“ zu finden.

[5] Die relevanten Stellen sind in dieser Arbeit in den Abschnitten „Aktualisierung der Dokumentation“ und „Händische Anpassung von HTML-Dateien“ zu finden.

- ☐ Wurden die Versionsnummern der RENEW-Plugins angepasst, sofern die jeweiligen Plugins verändert wurden? Dafür (mindestens):
  - Die Plugin-Versionsnummern überprüfen. Dafür z.B. die Versionsnummern, welche in den JAR-Dateinamen des aktuellen Release enthalten sind, mit denen aus dem vorherigen Release vergleichen.
  - Die im Handbuch referenzierten Plugin-Versionsnummern überprüfen. Dafür z.B. die Versionsnummern, welche in der `plugindefs.sty` aufgelistet werden, mit den Versionsnummern, welche in den JAR-Dateinamen des aktuellen Release enthalten sind, vergleichen.
- ☐ Wurde die geforderte Java-Version überall eingestellt und dokumentiert? Zu prüfen (mindestens):
  - Steht die korrekte Java-Version in der Build-Datei `build.gradle`?
  - Wird die korrekte Java-Version an den ensprechenden Stellen im Handbuch, in den Readme-Dateien und in den Dateien der Webseite gefordert?[6]

  Tip Die geforderte Java-Version sollte eine LTS-Version sein, so niedrig wie möglich und so hoch wie nötig gewählt werden. Nach Möglichkeit sollte diese Version im gesamten CI/CD-Prozess genutzt werden, d.h. in den Build-Dateien und den Docker-Images.
  Lt. CCPO sind in der CI/CD-Pipeline mittelfristig die jeweils neueste LTS-Version und optional bzw. händisch deren anschließende Versionen vorzusehen, damit frühzeitig veraltete Methoden (engl. deprecated methods) etc. erkannt und berücksichtigt werden können.

D.4 Handbuch und Readme-Dateien

- ☐ Wurden alle Informationen sowohl im Handbuch als auch in den Readme-Dateien (`README`, `README-macosx.txt` und README.md) aktualisiert?
- ☐ Sind nicht aktualisierte Informationen in Handbuch und Readme-Dateien noch aktuell?

D.5 Sourcecode

- ☐ Wurde das Source-Package auf verschiedenen Systemen kompiliert und gestartet?[7]
- ☐ Sind nur die Dateien enthalten, die auch veröffentlicht werden sollen?
  Dies sind für 4.x-Versionen i.d.R. im Unterordner `src`: der Quelltext der veröffentlichten Plugins, die Gradle-Dateien, die für das Kompilieren benötigt werden, und die JAR-Dateien der benötigten Bibliotheken im `lib`-Ordner. Und im Unterordner `doc`: die Dateien `README`, `LICENSE` und `COPYING`, das Handbuch als PDF und die TEX-Dateien des Handbuchs (`.tex`-, `.bib`-, `.sty`-Dateien und ein Ordner `images`).

[6] Die relevanten Stellen sind in dieser Arbeit im Abschnitt „Aktualisierung der Dokumentation“ für das Handbuch und die Readme-Dateien zu finden. Auf der Webseite wird die Java-Version in `install.html` und `download.html` im Versionsordner aufgeführt und sollte durch die JSON-Kontextdatei und die Templates korrekt aktualisiert worden sein.

[7] Ausführliche Anweisungen sind z.B. auf der Webseite in der `install.hmtl` zu finden unter „Install the Source package“.

D.6 Veröffentlichungsumfang

☐ Wurden nur die Plugins dem neuen Release hinzugefügt, die tatsächlich veröffentlicht werden sollen?

- RENEW umfasst wesentlich mehr Plugins als die, die veröffentlicht werden.
- Das Source-Package sollte nur den Quelltext der Plugins beinhalten, die innerhalb der verschiedenen Distribution-Packages und der zusätzlichen optionalen Plugins in ihren Binärversionen veröffentlicht werden.

☐ Schließt das neue Release Plugins ein, die nicht zum vorherigen Release[8] gehörten? Wenn ja:

- Ist das beabsichtigt?
- Wurde die richtige Veröffentlichungsform (Base-Package, anderes Distribution-Package, optionale Plugins, Extra-Plugins) für die erstmals veröffentlichten Plugins gewählt?

D.7 Webseite

☐ Wurden für die Erstellung der neuen Dateien für die Webseite die Skripte genutzt und die Anweisungen befolgt?

☐ Wurde darauf geachtet, dass die Einträge bzgl. des neuen Release (bspw. in den Dateien `history.html` und `issues.html`) konsistent zu den Einträgen vorheriger Releases ist?

☐ Stimmen die Informationen auf der Webseite mit denen im Handbuch überein?

☐ Wurden Dinge, wie die Release-Versionsnummer und das Veröffentlichungsdatum an den entsprechenden Stellen aktualisiert?

D.8 Allgemein

Wenn ein MAJOR-Release bevorsteht, müssen die Vorbereitungen besonders kritisch geprüft werden, insbesondere, falls sich Bauprozesse verändert haben. Hier bietet es sich an, Aspekte des neuen Release mit denen des vorigen zu vergleichen; bspw. welche Dateien in den unterschiedlichen Distribution-Packages sind und wie diese strukturiert sind. Wenn Unterschiede bestehen, muss hinterfragt werden, warum dies so ist.

[8] Die innerhalb von RENEW 2.6, 4.0 und 4.1 veröffentlichten Plugins sind in einer Übersicht in dieser Arbeit in Anhang B zu finden.

Literatur

Adams, Bram, Stephany Bellomo, Christian Bird, Tamara Marshall-Keim, Foutse Khomh und Kim Moir (2015). "The Practice and Future of Release Engineering: A Roundtable with Three Release Engineers". In: *IEEE Software* 32.2, S. 42–49. DOI: 10.1109/MS.2015.52.

Adams, Bram, Christian Bird, Foutse Khomh und Kim Moir (2013). "1st International Workshop on Release Engineering (RELENG 2013)". In: *2013 35th International Conference on Software Engineering (ICSE)*, S. 1545–1546. DOI: 10.1109/ICSE.2013.6606779.

Aggarwal, Anirudh Kumar und V. S. Mani (2019). "Using product line engineering in a globally distributed agile development team to shorten release cycles effectively". In: *Proceedings of the 14th International Conference on Global Software Engineering*. ICGSE '19. Montreal, Quebec, Canada: IEEE Press, S. 48–51. DOI: 10.1109/ICGSE.2019.000-7. URL: https://doi.org/10.1109/ICGSE.2019.000-7.

Al Shriteh, Youssef (Nov. 2022). "Qualitätssicherung der Software Renew durch automatisierte Tests und Testumgebung". Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Anonymous Author(s) (2024). "Tackling Project-Based Learning in a Large-Scale Educational Project: An Experience Report." In: *In Proceedings of 46th International Conference on Software Engineering (ICSE 2024)*.

Barnett, Taylor (2020). *Documentation as a Path to DevOps Automation*. https://www.transposit.com/devops-blog/devops/2020.04.02-documentation-as-a-path-to-devops-automation/. [Zuletzt aufgerufen: 06.04.2024].

Beneken, Gerd Hinrich, Felix Hummel und Martin Kucich (2022). *Grundkurs agiles Software-Engineering Ein Handbuch für Studium und Praxis*. 1st ed. Springer Fachmedien Wiesbaden. ISBN: 978-3-658-37371-9.

Billigmann, Sven (2023). "Der Release einer Renew Version auf der Renew Website". Projektbericht AOSE22. URL: https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?spaceKey=AOSE22&title=Abgabe+Berichte&preview=/325353561/353763357/9billigm.pdf.

Brun, Yuriy, Reid Holmes, Michael D. Ernst und David Notkin (2011). "Proactive Detection of Collaboration Conflicts". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, S. 168–178. ISBN: 9781450304436. DOI: 10.1145/2025113.2025139. URL: https://doi.org/10.1145/2025113.2025139.

Bryant, Daniel und Abraham Marín-Pérez (2018). *Continuous Delivery in Java: Essential Tools and Best Practices for Deploying Code to Production*. First edition. O'Reilly Media. ISBN: 9781491985991. URL: https://www.gbv.de/dms/tib-ub-hannover/1042083010.pdf.

Cabac, Lawrence, Michael Haustermann und David Mosteller (2016). “Renew 2.5 - Towards a Comprehensive Integrated Development Environment for Petri Net-Based Applications”. In: *Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*. Hrsg. von Fabrice Kordon und Daniel Moldt. Bd. 9698. Lecture Notes in Computer Science. Springer-Verlag, S. 101–112. ISBN: 978-3-319-39085-7. URL: http://dx.doi.org/10.1007/978-3-319-39086-4_7.

Chacon, Scott und Ben Straub (2014). “Pro Git”. In: 2nd. Berkeley, CA: Apress, S. 1–13. ISBN: 978-1-4842-0076-6. DOI: 10.1007/978-1-4842-0076-6_1. URL: https://doi.org/10.1007/978-1-4842-0076-6%5C_1.

Chen, Lianping (2015). “Continuous Delivery: Huge Benefits, but Challenges Too”. In: *IEEE Software* 32.2, S. 50–54. DOI: 10.1109/MS.2015.27.

Clasen, Laif-Oke (März 2023). “Untersuchung von Softwarearchitektur und Projektorganisation unter Berücksichtigung der Wechselwirkungen im Kontext von Lehrprojekten am Beispiel verteilter Entwicklung der Open-Source Software Renew”. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Clasen, Laif-Oke und Marcel Hansson (2023). *Paose: Code und Thesis Checkliste*. https://git.informatik.uni-hamburg.de/tgi/theses/paose-code-and-thesis-checkliste. [Zuletzt aufgerufen: 31.01.2024].

Clasen, Laif-Oke, Daniel Moldt, Marcel Hansson, Sven Willrodt und Lukas Voß (2022). “Enhancement of Renew to Version 4.0 using JPMS”. In: *Proceedings of the International Workshop on Petri Nets and Software Engineering 2022 co-located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022*. Hrsg. von Michael Köhler-Bußmeier, Daniel Moldt und Heiko Rölke. Bd. 3170. CEUR Workshop Proceedings. CEUR-WS.org, S. 165–176. URL: https://ceur-ws.org/Vol-3170.

Daschkewitsch, Arkadij (2019). “Modularisierung des Renew-Plugin Systems”. Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Dwyer, Jack (2024). *Complete 3-Step Guide On Automated Release Management (& 20 Best Practices)*. https://zeet.co/blog/automated-release-management. [Zuletzt aufgerufen: 06.04.2024].

Dyck, Andrej, Ralf Penners und Horst Lichter (2015). “Towards Definitions for Release Engineering and DevOps”. In: *Proceedings of the Third International Workshop on Release Engineering*. RELENG '15. Florence, Italy: IEEE Press, S. 3.

Feldmann, Matthias (Nov. 2019). “Containerization of the Reference Net Workshop and Evaluation of Interprocess Communication Technologies for Containerized Multi-Agent Net Applications”. Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Feller, Joseph und Brian Fitzgerald (2002). *Understanding Open Source Software Development*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-73496-6.

Fitz, Timothy (2009). *Continuous Deployment for Downloadable Client Software*. http://timothyfitz.com/2009/03/09/cd-for-client-software/. [Zuletzt aufgerufen: 27.03.2024].

Fowler, Martin (2011). *Frequency Reduces Difficulty*. https://martinfowler.com/bliki/FrequencyReducesDifficulty.html. [Zuletzt aufgerufen: 18.04.2024].

— (2013). *Continuous Delivery*. https://martinfowler.com/bliki/ContinuousDelivery.html. [Zuletzt aufgerufen: 04.04.2024].

— (2024). *Continuous Integration*. https://www.martinfowler.com/articles/continuousIntegration.html. [Zuletzt aufgerufen: 04.04.2024].

GitLab Inc. (2024a). *Caching in GitLab CI/CD*. https://docs.gitlab.com/ee/ci/caching/. [Zuletzt aufgerufen: 03.04.2024].

— (2024b). *CI/CD pipelines*. https://docs.gitlab.com/ee/ci/pipelines/. [Zuletzt aufgerufen: 03.04.2024].

— (2024c). *CI/CD YAML syntax reference*. https://docs.gitlab.com/ee/ci/yaml/. [Zuletzt aufgerufen: 03.04.2024].

— (2024d). *CI/CD YAML syntax reference*. https://docs.gitlab.com/ee/ci/yaml/index.html/. [Zuletzt aufgerufen: 03.04.2024].

— (2024e). *Customize pipeline configuration*. https://docs.gitlab.com/ee/ci/pipelines/settings.html. [Zuletzt aufgerufen: 03.04.2024].

— (2024f). *Docker executor*. https://docs.gitlab.com/runner/executors/docker.html. [Zuletzt aufgerufen: 03.04.2024].

— (2024g). *GitLab CI/CD variables*. https://docs.gitlab.com/ee/ci/variables/. [Zuletzt aufgerufen: 03.04.2024].

— (2024h). *GitLab Runner*. https://docs.gitlab.com/runner/. [Zuletzt aufgerufen: 03.04.2024].

— (2024i). *Job artifacts*. https://docs.gitlab.com/ee/ci/jobs/job_artifacts.html. [Zuletzt aufgerufen: 03.04.2024].

— (2024j). *Jobs*. https://docs.gitlab.com/ee/ci/jobs/. [Zuletzt aufgerufen: 03.04.2024].

— (2024k). *Merge conflicts*. https://docs.gitlab.com/ee/user/project/merge_requests/conflicts.html. [Zuletzt aufgerufen: 06.03.2024].

— (2024l). *Merge request approvals*. https://docs.gitlab.com/ee/user/project/merge_requests/approvals/. [Zuletzt aufgerufen: 06.03.2024].

— (2024m). *Merge request workflows*. https://docs.gitlab.com/ee/user/project/merge_requests/authorization_for_merge_requests.html. [Zuletzt aufgerufen: 06.03.2024].

— (2024n). *Merge requests*. https://docs.gitlab.com/ee/user/project/merge_requests/. [Zuletzt aufgerufen: 06.03.2024].

— (2024o). *Squash and merge*. https://docs.gitlab.com/ee/user/project/merge_requests/squash_and_merge.html. [Zuletzt aufgerufen: 04.04.2024].

Google LLC (2015). *Documentation Best Practices*. https://google.github.io/styleguide/docguide/best_practices.html. [Zuletzt aufgerufen: 08.04.2024].

Gradle Inc. (2023a). *Command-Line Interface Reference*. https://docs.gradle.org/current/userguide/command_line_interface.html. [Zuletzt aufgerufen: 15.12.2023].

Gradle Inc. (2023b). *Gradle Wrapper Reference*. https://docs.gradle.org/current/userguide/gradle_wrapper.html. [Zuletzt aufgerufen: 15.12.2023].

Hansson, Marcel (2020). "Konsolidierung workflowbezogener Renew-Plugins". Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Heinze, Alexander (März 2022). "Umstellung des Quellcodes der Software Renew auf eine Multi-Repository-Struktur unter Verwendung der Abhängigkeitsmanagement-Funktionalität von Gradle". Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Hoffmann, Michael (2020). *How To Automatically Generate A Helpful Changelog From Your Git Commit Messages*. https://mokkapps.de/blog/how-to-automatically-generate-a-helpful-changelog-from-your-git-commit-messages. [Zuletzt aufgerufen: 18.03.2024].

Humble, Jez (2017a). *Continuous Delivery*. https://continuousdelivery.com/. [Zuletzt aufgerufen: 04.04.2024].

— (2017b). *Continuous Integration*. https://continuousdelivery.com/foundations/continuous-integration/. [Zuletzt aufgerufen: 04.04.2024].

Humble, Jez und David G. Farley (2011). *Continuous delivery [reliable software releases through build, test, and deployment automation]*. The Addison-Wesley signature series: A Martin Fowler signature book. Addison-Wesley. ISBN: 978-0-321-60191-9.

Janneck, Jan Robert (März 2021). "Modularizing a Plugin System Using Java Modules: Application to a Medium-Sized Open-Source Project". Masterarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Jiang, Huaxi, Jie Zhu, Li Yang, Geng Liang und Chun Zuo (2022). *DeepRelease: Language-agnostic Release Notes Generation from Pull Requests of Open-source Software*. arXiv: 2201.06720 [cs.SE].

Johnsen, Jonte (März 2022). "Sicherung der Lauffähigkeit von Renew 4.0 bei einer Umstellung der Java LTS Version". Bachelorarbeit. Vogt-Kölln Str. 30, D-22527 Hamburg: Universität Hamburg, Fachbereich Informatik.

Jørgensen, Niels (2001). "Putting it all in the trunk: Incremental software development in the FreeBSD open source project". In: *Information Systems Journal* 11.4, S. 321–336.

Karvonen, Teemu, Woubshet Behutiye, Markku Oivo und Pasi Kuvaja (2017). "Systematic literature review on the impacts of agile release engineering practices". In: *Information and Software Technology* 86, S. 87–100. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.01.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584917300678.

Kummer, Olaf und Frank Wienberg (März 1999). *Renew -- User Guide*. Release 1.0. Available at: http://www.renew.de/. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group. Hamburg. URL: http://www.renew.de/.

Kummer, Olaf, Frank Wienberg, Michael Duvigneau, Lawrence Cabac, Michael Haustermann und David Mosteller (Apr. 2022). *Renew -- The Reference Net Workshop*. Release 2.6. URL: http://www.renew.de/.

— (Feb. 2023a). *Renew -- The Reference Net Workshop*. Release 4.1. URL: http://www.renew.de/.

— (Feb. 2023b). *Renew -- User Guide (Release 4.1)*. Release 4.1. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group. Hamburg. URL: http://www.renew.de/.

Langemann, Nicolas (2023). *Software-Testing mit ChatGPT als Assistent*. https://heise.de/-9351842. [Zuletzt aufgerufen: 12.03.2024].

Menezes, José William, Bruno Trindade, João Felipe Pimentel, Tayane Moura, Alexandre Plastino, Leonardo Murta und Catarina Costa (2020). "What Causes Merge Conflicts?" In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. SBES '20. Natal, Brazil: Association for Computing Machinery, S. 203–212. ISBN: 9781450387538. DOI: 10.1145/3422392.3422440. URL: https://doi.org/10.1145/3422392.3422440.

Michlmayr, Martin, Francis Hunt und David Probert (Juni 2007). "Release Management in Free Software Projects: Practices and Problems". In: Hrsg. von Joseph Feller, Brian Fitzgerald, Walt Scacchi und Alberto Sillitti. Bd. 234. Boston, MA: Springer US, S. 295–300. ISBN: 978-0-387-72485-0. DOI: 10.1007/978-0-387-72486-7_31.

Moldt, Daniel, Jonte Johnsen, Relana Streckenbach, Laif-Oke Clasen, Michael Haustermann, Alexander Heinze, Marcel Hansson, Matthias Feldmann und Karl Ihlenfeldt (2023a). "RENEW: Modularized Architecture and New Features". In: *Application and Theory of Petri Nets and Concurrency*. Hrsg. von Robert Gomes Luis and Lorenz. Cham: Springer Nature Switzerland, S. 217–228. ISBN: 978-3-031-33620-1.

— (2023b). "RENEW: Modularized Architecture and New Features". In: *Application and Theory of Petri Nets and Concurrency - 44th International Conference, PETRI NETS 2023, Lisbon, Portugal, June 25-30, 2023, Proceedings*. Hrsg. von Luís Gomes und Robert Lorenz. Bd. 13929. Lecture Notes in Computer Science. Cham, Switzerland: Springer Nature Switzerland AG, S. 217–228. DOI: 10.1007/978-3-031-33620-1_12. URL: https://doi.org/10.1007/978-3-031-33620-1_12.

Nath, Sristy Sumana und Banani Roy (Juli 2021). "Towards Automatically Generating Release Notes using Extractive Summarization Technique". In: *International Conferences on Software Engineering and Knowledge Engineering*. SEKE2021. KSI Research Inc. DOI: 10.18293/seke2021-119. URL: http://dx.doi.org/10.18293/SEKE2021-119.

Nelson, Nicholas, Caius Brindescu, Shane McKee, Anita Sarma und Danny Dig (2019). "The life-cycle of merge conflicts: processes, barriers, and strategies". In: *Empirical Software Engineering*, S. 1–44.

Poo-Caamaño, Germán (2016). "Release Management in Free and Open Source Software Ecosystems". Dissertation. University of Victoria, Canada. URL: http://hdl.handle.net/1828/7648.

Preißel, René und Bjørn Stachmann (2019). *Git: dezentrale Versionsverwaltung im Team: Grundlagen und Workflows*. 5., aktualisierte und erweiterte Auflage. dpunkt.verlag. ISBN: 978-3-86490-649-7.

Preston-Werner, Tom (2013). *Semantic Versioning*. web. URL: http://semver.org/.

Rahman, Md Tajmilur (Apr. 2015). "Investigating modern release engineering practices". In: S. 607–608. DOI: 10.1109/SANER.2015.7081893.

Rashid, Mehvish, Paul M. Clarke und Rory V. O'Connor (2019). "A systematic examination of knowledge loss in open source software projects". In: *International Journal of Information Management* 46, S. 104–123. ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2018.11.015. URL: https://www.sciencedirect.com/science/article/pii/S0268401217310095.

Reed, J. Paul (2020). *When /bin/sh Attacks: Revisiting „Automate All the Things"*. https://www.usenix.org/conference/srecon20americas/presentation/reed. [Zuletzt aufgerufen: 06.04.2024].

Renew-Development-Team (2023a). *Einführung und Basics*. https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?pageId=497811507. [Zuletzt aufgerufen: 10.04.2024].

— (2023b). *Git*. https://tgipm.informatik.uni-hamburg.de/confluence/display/RENEWDEV/Git. [Zuletzt aufgerufen: 28.01.2024].

— (2023c). *Sonargraph (german)*. https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?pageId=227311622. [Zuletzt aufgerufen: 10.04.2024].

— (2024). *Code Review Checkliste*. https://tgipm.informatik.uni-hamburg.de/confluence/display/RENEWDEV/Code+Review+Checkliste. [Zuletzt aufgerufen: 28.01.2024].

Rouse, Margaret (2017). *Technical Debt*. https://www.techopedia.com/definition/27913/technical-debt. [Zuletzt aufgerufen: 10.04.2024].

Rowell (2023). *SaaS vs Non-SaaS? What's The Difference*. https://saaslucid.com/saas-vs-non-saas/. [Zuletzt aufgerufen: 26.03.2024].

Schuhardt, Jannis (2023). "Renew-Web: Automating the Renew releaseprocess using GitLab CI/CD". Projektbericht AOSE22. URL: https://tgipm.informatik.uni-hamburg.de/confluence/pages/viewpage.action?spaceKey=AOSE22&title=Abgabe+Berichte&preview=/325353561/381222940/0schuhar.pdf.

Schwaber, Ken und Jeff Sutherland (2020). "The 2020 Scrum Guide". In: [Zuletzt aufgerufen: 22.11.2023].

Scrum.org (2020). *The Scrum Framework Poster*. https://www.scrum.org/resources/scrum-framework-poster. [Zuletzt aufgerufen: 22.11.2023].

Shake Technologies, Inc. (2023). *Alpha vs beta apps and nightly vs. production builds*. https://www.shakebugs.com/blog/alpha-vs-beta-apps/. [Zuletzt aufgerufen: 18.02.2024].

Shringi, Kanchan und Rishi Singh (Feb. 2024). *SE Radio 603: Rishi Singh on Using GenAI for Test Code Generation*. IEEE Software. Software Engineering Radio. URL: https://se-radio.net/2024/02/se-radio-603-rishi-singh-on-using-genai-for-test-code-generation/.

Silva, Antonio César Brandão Gomes da, Glauco de Figueiredo Carneiro, Antonio Carlos Marcelino de Paula, Miguel Pessoa Monteiro und Fernando Brito e Abreu (2016). "Agility and Quality Attributes in Open Source Software Projects Release Practices". In: *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, S. 107–112. DOI: 10.1109/QUATIC.2016.029.

Slimmon, Dan (2019). *Do-nothing scripting: the key to gradual automation*. https://blog.danslimmon.com/2019/07/15/do-nothing-scripting-the-key-to-gradual-automation/. [Zuletzt aufgerufen: 06.04.2024].

Sommerville, Ian (2018). *Software Engineering*. Hrsg. von Katharina Pieper und Petra Alm. 10., aktualisierte Auflage. it - informatik. Pearson. ISBN: 978-3-86326-835-0. URL: https://elibrary.pearson.de/book/99.150005/9783863268350.

Spinellis, Diomidis (2018). "The Challenges and Practices of Release Engineering". In: *IEEE Software* 35.2, S. 4–7. DOI: 10.1109/MS.2018.1661312.

The Open Source Initiative (2023a). *OSI Approved Licenses*. https://opensource.org/licenses/. [Zuletzt aufgerufen: 12.11.2023].

— (2023b). *The Open Source Definition*. https://opensource.org/osd/. [Zuletzt aufgerufen: 09.11.2023].

The YAML Project (2021). *The Official YAML Web Site*. https://yaml.org/. [Zuletzt aufgerufen: 02.12.2023].

Thönes, Johannes und Jez Humble (Feb. 2015). *SE Radio 221: Jez Humble on Continuous Delivery*. IEEE Software. Software Engineering Radio. URL: https://se-radio.net/2015/02/episode-221-jez-humble-on-continuous-delivery/.

Tsay, Jason, Hyrum K. Wright und Dewayne E. Perry (2011). "Experiences mining open source release histories". In: *Proceedings of the 2011 International Conference on Software and Systems Process*. ICSSP '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, S. 208–212. ISBN: 9781450307307. DOI: 10.1145/1987875.1987911. URL: https://doi.org/10.1145/1987875.1987911.

Ubuntu Manpage Repository (2019). *Ubuntu Manpage: xvfb-run*. https://manpages.ubuntu.com/manpages/trusty/man1/xvfb-run.1.html. [Zuletzt aufgerufen: 03.04.2024].

Witko, Kinga (2019). *What Are The Benefits of Having Nightly Builds*. https://blog.testproject.io/2019/10/14/what-are-the-benefits-of-having-nightly-builds/. [Zuletzt aufgerufen: 18.02.2024].

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

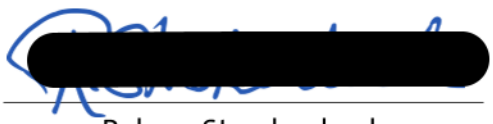
Hamburg, den 28. Mai 2024

Relana Streckenbach

Erklärung zur Veröffentlichung

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 28. Mai 2024



Relana Streckenbach