Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Seminararbeit

Release-Management

Softwareentwicklung in der Wissenschaft

Sebastian Schulz

28. März 2011

Zusammenfassung

In dieser Seminararbeit geht es um das Thema Release-Management. Release-Management beschreibt den Prozess von der Softwareentwicklung bis zur Veröffentlichung der Software. Dabei stellt sich die Frage, wie man die Qualität der Software verbessern und wie man den Ablauf organisieren kann. Anhand von Beispielen und bewährten Vorgehensweisen wird beschrieben, wie man in eigenen Softwareprojekten Release-Management nutzen kann. Freie Softwareprojekte und deren Vorgehensweisen werden beleuchtet um an praktische Beispiele zu lernen. Es wird heraus gearbeitet, welchen Mehrwert die Wissenschaft haben kann und wie man ohne großen Aufwand Release-Management optimieren kann.

Inhaltsverzeichnis

1 Einleitung

In meiner Seminararbeit möchte ich mich mit Release-Management beschäftigen. Verschiedene Begriffe werden mal Release-Management zugeschrieben und mal anderen Themengebieten zugeordnet. Daher möchte ich in Kapitel 2 unter anderem Release-Management näher definieren und einordnen.

Um mich dem Thema zu nähern, habe ich mir überlegt wie ich bei einem eigenen Softwareprojekt vorgehen würde. Dabei bin ich auf folgende Fragen gekommen. Was sind die Anforderungen? Und wie sind die Anforderungen an ein Release-Management in der Wissenschaft? Nun aber zu meinem Aufbau der Seminararbeit.

In Kapitel 3 möchte ich den Prozess beschreiben, den man durchläuft wenn man eine Software veröffentlichen möchte. Es gibt dabei naturgemäß nicht den optimalen Weg. Jedoch starte ich einen Versuch mögliche Vorgehensmethoden zu beschreiben. Kleine Beispiele sollen dabei helfen, wie man Release-Management in eigenen Projekten nutzen kann. Ich werde unter anderem auf Versionierung zu sprechen kommen und wie man mit einfachen Mitteln auch in kleinen Projekten Release-Management nutzen kann.

Meine Idee war, dass ich mir Projekte aus der freien Software-Szene anschaue. Sie bieten einen guten Einblick in ihr jeweiliges Release-Management. In Kapitel 4 betrachte ich drei Projekte im Detail. Jedes dieser freien Softwareprojekten hat einen anderen Ansatz bei diesem Thema. Sie können für das eigene Vorgehen als gutes Vorbild dienen.

Es stellt sich die Frage, welchen Nutzen Release-Management der Wissenschaft bringen kann. Den Bezug zur Softwareentwicklung in der Wissenschaft möchte ich in Kapitel 5 herstellen.

In meiner Seminararbeit werde ich ab und zu Begriffe verwenden, die aus dem Bereich „Projektmanagement“ und dem Bereich „agile Softwareentwicklung“ bekannt sind. Ich werde diese Begriffe nicht tiefergehend behandeln. Eine Aufwandsabschätzung habe ich ebenso nicht betrieben. Auch nicht betrachtet habe ich Projekt-Management Software und das Thema Bug-Tracking. Es spielt im Bereich der Qualitätssicherung durchaus eine wichtige Rolle. Mir war wichtiger in Erfahrung zu bringen, ob es einen gangbaren, einfachen Weg gibt, Release-Management zu realisieren.

2 Definition

Release-Management ist ein sehr weitläufiger Begriff. Je nach Definition wird unter dem Begriff etwas anderes verstanden. Daher halte ich es für notwendig den Begriff „Release-Management“ näher zu definieren und die Aufgaben zu beschreiben. Wie ich in der Abbildung 1 dargestellt habe, besteht eine Beziehung zwischen Konfigurationsmanagement und Änderungsmanagement. Weiter besteht eine Beziehung zwischen Release-Management und den zwei Management-Disziplinen.

In diesem Kapitel möchte die drei Management-Disziplinen näher definieren und eingrenzen und erläutern welche Aufgaben sie haben.

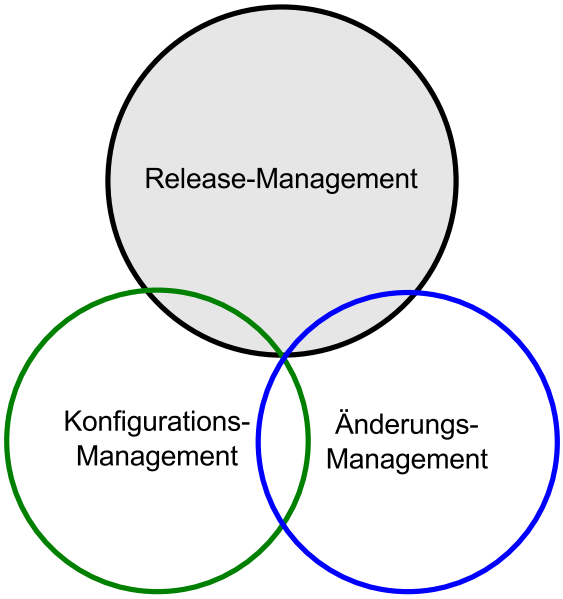

Abbildung 1: Zusammenhang von Release-, Änderungs- und Konfigurationsmanagement

2.1 Was ist Release-Management?

Van der Hoek definiert Release-Management als „Prozess, durch den Software zur Verfügung gestellt wird.“ [HW03]. In meinen Augen beschreibt diese Definition gut den Aufgabenbereich von Release-Management.

Die wichtigsten Aufgaben von Release-Management sind die Festlegung des Funktionsumfangs, die Festlegung des Zeitplans, die Qualitätssicherung, die Dokumentation des Umfangs und der Änderung der Software und die Sicherstellung der Reproduzierbarkeit [Wik11a]. Die Qualitätssicherung soll dazu beitragen, dass Kriterien und Leitfäden eingehalten werden.

2.2 Was ist Änderungsmanagement?

In der ISO-Norm 20000-1 wird Änderungsmanagement wie folgt definiert: „Änderungsmanagement stellt sicher, dass alle Änderungen bewertet, erprobt, implementiert und überprüft werden.“ [ISO05]. Die Aufgaben von Änderungsmanagement sind, dass Änderungen initiiert, dokumentiert und autorisiert werden. Es soll dazu beitragen, dass Änderungen und davon abhängige Risiken abschätzbar werden. Weitere Aufgaben von Änderungsmanagement sind die Koordinierung von Implementierungen und die abschließende Überprüfung derer.

Wie wir sehen, kann man Änderungsmanagement als Teilmenge von Release-Management sehen. Bei einem Prozess zur Freigabe einer Software ist es natürlich auch wichtig Änderungen zu dokumentieren und zu überprüfen. Es ist nötig um ein Mindestmaß an Qualitätssicherung zu erreichen.

2.3 Was ist Konfigurationsmanagement?

„Konfigurationsmanagement koordiniert Maßnahmen um Konfigurationen zu lenken und zu kontrollieren“ [ISO04]. Man versteht unter Konfiguration die Anpassung an ein System, auf dem eine Software laufen soll. Die Aufgaben von Konfigurationsmanagement sind das Definieren und das Verfolgen von Prozessen, Vorgänge zu dokumentieren, Versionierung und Konfliktbehandlung sowie das Verwalten der Voraussetzungen einer Software.

Das Verfolgen der Prozesse und eine Versionierung helfen ein gutes Release-Management zu realisieren. Gerade die Versionierung halte ich für sehr wichtig. Wir sehen auch hier, dass es eine große Schnittmenge mit Release-Management gibt.

3 Release-Management

Dieses Kapitel soll Release-Management näher beschreiben. Anhand von Beispielen und bewährten Praktiken möchte ich verschiedene Vorgehensmethoden aufzeigen. Jede Methode stellt eine Möglichkeit dar, den Release-Prozess zu optimieren.

3.1 Release-Manager

Der Release-Manager ist die verantwortlich Person und hat zur Aufgabe, die den Release-Prozess zu koordinieren. Er legt den zeitlichen Fahrplan fest und ist die letzte Entscheidungsinstanz [Ere03]. Er entscheidet letztendlich darüber, welche neuen Funktionen in das Release einfließen und welche davon zu einem späteren Zeitpunkt implementiert werden. Er entscheidet auch darüber inwiefern die veröffentlichte Version als stabil gekennzeichnet wird.

Bei großen Projekten ist es denkbar, dass die Rolle des Release-Managers von mehreren Personen geteilt wird. Es ist häufig auch denkbar, dass Aufgaben delegiert werden. So kann es sinnvoll sein die Pflege der Dokumentation oder die Pflege der Release-Notes an Personen im Entwicklerteam als Aufgabe zu übertragen.

Es gibt verschiedene Möglichkeiten den Release-Manager zu bestimmen. In freien Software-Projekten kann es üblich sein, dass hierzu eine Wahl stattfindet [The11a]. In jedem Fall sollte ein Release-Manager ein Mindestmaß an Erfahrung in Software-Entwicklung mitbringen.

3.2 Der Weg zur Freigabe

Bevor wir eine Software freigeben können, ist es notwendig die Abläufe vor der Freigabe zu planen. So sollte der zeitliche Ablauf und Fristen festgelegt werden. Im Projekt sollten Meilensteine festgelegt werden, in denen geschrieben steht, welche Funktionen in die Software implementiert werden sollen. Nach der Veröffentlichung ist es wichtig die gewonnenen Erfahrungen zurückfließen zu lassen. Es soll hierdurch gewährleistet werden, dass Fehler vermieden und zeitliche Verzögerungen zukünftig besser eingeschätzt werden können. Hilfreich kann eine Checkliste sein, die während dem Veröffentlichungsprozess abgearbeitet wird. Die gewonnen Erfahrungen können abschließend die Checkliste ergänzen.

Test vor der Veröffentlichung

Vor einer Veröffentlichung ist es notwendig die Software ausführlich zu testen. Viele Projekte haben Kriterien, die erfüllt werden müssen bevor die Software veröffentlicht werden darf [Ere03].

Nachdem neue Funktionen implementiert wurde, bietet sich an, die Schnittstellen der Software bis zur Freigabe einzufrieren. Im weiteren Verlauf sollen also ausschließlich Fehler behoben werden. Dem erfahrenen Endbenutzer bietet man hierdurch die Möglichkeit an mit der Software zu experimentieren. Externe Entwickler, welche die Software für eigene Projekte nutzen, können ihre eigenen Schnittstellen an die Software anpassen. Durch Veröffentlichungen von Patches bietet man externen Entwicklern kleine Verbesserungen der Software an. So wird die Software bis zur Freigabe Stück für Stück gehärtet.

Viele freie Software-Projekte haben automatisierte Testabläufe etabliert. Eine Möglichkeit ist, die Software auf unterschiedlichen Plattformen übersetzen zu lassen. Eine andere Möglichkeit ist ein Testszenario für jeden gefunden Fehler zu entwerfen. So vermeidet man schon frühzeitig das Fehler womöglich wiederkehren.

Freigabe und Abnahme

Der Release-Manager entscheidet letztlich darüber ob und wann eine Software freigegeben werden kann. Jedoch ist es auch denkbar, das eine unabhängige Gruppe diese Entscheidung übernimmt. Bei einem Projekt mit mehreren Release-Managern kann eine Freigabe der Software erfolgen, wenn ein vorher definierter Konsens zwischen den Managern besteht.

Die Entscheidung für eine Freigabe sollte anhand vorher definierte Kriterien getroffen werden. So sollte als Kriterium festgelegt werden, dass automatisierte Testabläufe fehlerfrei vonstatten gehen.

Bei einer Freigabe sollte notiert werden, welche Abhängigkeiten bestehen und wie die Installation abläuft. Dazu bietet es sich an eine Release-Note auf der Website bereitzustellen und als Textdatei der Software beizufügen. Eine Ankündigung sollte in Form einer E-Mail an eine Mailingliste oder einem Eintrag auf einer Website erfolgen.

Der Zugriff auf die Software kann über einen Webserver und Spiegelserver erfolgen. Spiegelserver haben die Aufgabe den Webserver zu entlasten. Gerade in den ersten Stunden passiert es häufig, dass die Hauptserver wegen der vielen Downloads zusammenbrechen [Her08].

Archivierung der Software

Ziel soll es sein, den Aufwand für den Endbenutzer möglichst gering zu halten. Je nach Benutzergruppe soll erreicht werden, dass möglichst viele Benutzer mit dem Format der Archivierung umgehen können. Das unter Unix-Nutzer übliche Tar-Gzip-Archiv ist beispielsweise für einen Windows-Nutzer nur mit einem höheren Aufwand nutzbar. Dem Archiv sollte eine Beschreibung bei liegen wie die Software installiert werden kann und welche anderen Softwarepakete benötigt werden. Häufig findet man Dateien `README` und `Changelog.txt` die den Endbenutzer darüber informieren wie das ganze nutzbar ist und was sich im Vergleich zu älteren Versionen geändert hat.

Ein weiteres Ziel soll sein, dass die Versionen langfristig verfügbar sind. Gerade ältere Versionen sollen, parallel zur aktuellen Version, angeboten werden. Dadurch ist es möglich Fehler zu reproduzieren und Verhalten der unterschiedlichen Versionen zu vergleichen.

Es sollte festgelegt werden, welche Archiv-Formate bereitgestellt werden. Ist es beispielsweise sinvoll fertig kompilierte Binärpakete bereit zustellen? Für welche Plattformen? Es bietet sich an, diese Fragen in einem Leitfaden zu dokumentieren. Letztendlich ist es eine Frage der Ressourcen. Entwickelt man für genau eine Plattform, ist der Aufwand geringer und die Frage schnell beantwortet.

Eine Prüfsumme der veröffentlichten Dateien und eine Signierung durch den Release-Manager sollte in Erwägung gezogen werden. Es garantiert die Integrität der Software und verhindert, dass Malware oder ähnliches zum Endbenutzer gelangen.

Umgebungen und Installation

Um eine bessere Qualität der Software zu bekommen, bietet es sich an verschiedene Umgebungen zu etablieren. Denkbar wäre neben einer Produktionsumgebung und einer Entwicklungsumgebung, eine Testumgebung die ausschließlich zur Fehlersuche da ist. Große Software-Firmen benutzen eine weitere Umgebung wie zum Beispiel „Vorschau“. Auf dieser Umgebung haben ausgewählte und erfahrene Endbenutzer die Möglichkeit frühzeitig Erfahrung mit neuen Funktionen zu sammeln.

In einem Cluster stellt sich die Frage, wie die Software installiert werden kann ohne das der laufende Betrieb gestört wird und das bei möglichst geringem Aufwand. Daher möchte ich das Thema „Softwareverteilung (Deployment)“ an dieser Stelle erwähnen. Softwareverteilung beschäftigt sich mit der Inbetriebnahme von Software auf Anwender-PCs und Servern [Wik11c].

Viele notwendige Schritte während einer Installation lassen sich mit einem Build-Skript ab arbeiten. Jedoch gibt es immer „Handarbeit“ zu erledigen. So kann es notwendig sein, dass Datenbanken oder Konfigurationen händisch angepasst werden müssen. Durch eine Auflistung der Anpassungen ist es unter Umständen möglich, bei späteren Veröffentlichungen das Build-Skript zu verbessern.

3.3 Versionierung

Die meisten Software-Entwicklung arbeiten an unterschiedlichen Versionen ihrer Anwendung [CSK02]. Es besteht daher der Wunsch diese Versionen eindeutig unterscheiden zu können. Aus diesem Grund ordnet man den verschiedenen Veröffentlichungen Versionsnummern zu.

Eine Versionierung der Software bietet die Möglichkeit den Stand der Entwicklung zu kennzeichnen. So kann man an der Versionsnummer erkennen in welchem Zustand sich die jeweilige Version befindet. Der Endbenutzer kann so besser entscheiden, welche Version für ihn den meisten Mehrwert bringt. Im Gegensatz zu einem Endbenutzer nimmt ein Entwickler in Kauf, dass die Software noch fehlerbehaftet ist und noch nicht ausführlich getestet wurde. In der Regel wird einem Release eine Versionsnummer zugeordnet. Anhand der Versionsnummer ist erkennbar, wie stabil die Version ist.

Eine Versionsverwaltung, wie zum Beispiel Git, bieten Unterstützung zur Versionierung von Software. So ist es möglich Meilensteine als Schlagwörter zu definieren. Im Listing 1 habe ich ein einfaches Beispiel beigefügt. Es soll aufzeigen, wie man in Git Schlagwörter verwalten kann. In der ersten Zeile füge ich ein Schlagwort mit dem Namen „v1.1beta“ dem Repository hinzu. In der zweiten Zeile lasse ich mir sämtliche Schlagwörter auflisten. In unserem Falle listet Git das eben angelegte Schlagwort auf. In der vierten Zeile lösche ich schließlich unser Schlagwort und bekommen von Git eine Rückmeldung, dass dieser Vorgang erfolgreich war.

Listing 1: Beispiel für Schlagwörter in Git

```
1 max@muster ~/repo$ git tag -a "v1.1beta"
2 max@muster ~/repo$ git tag -l
3 v1.1beta
4 max@muster ~/repo$ git tag -d "v1.1beta"
5 Deleted tag 'v1.1beta' (was e4ee032)
```

Git und auch andere Versionsverwaltungen bieten die Möglichkeit in unterschiedlichen Zweigen (Branching) zu entwickeln. So ist es denkbar, dass man in einem Zweig eine stabile Version verwaltet. In einem anderen Zweig wird die eigentliche Weiterentwicklung vorangetrieben. Zur tieferen Betrachtung von Git möchte ich auf die Seminararbeit von Robert Wiesner verweisen [Wie11].

Welche Schemata gibt es für Versionierung? Die gebräuchlichste Möglichkeit ist eine iterierbare Abfolge. Dabei können Zahlen (0.9, 1.0, 1.2) zum Einsatz kommen. Wahlweise

kann auch eine alphabetische Anordnung genutzt werden. Es ist auch üblich beide Varianten zu mischen (1.1a, 1.1b, 1.2c). Es ist denkbar die Versionsnummer in Gruppen zu Gliedern.

Als Beispiel nehmen ich mal die Gliederung `Major.Minor[.Revision]` an. `Major` würde inkrementiert werden, wenn es starke Änderungen an der Software gibt. So ist es üblich, dass man `Major` erhöht, wenn sich die Schnittstellen der Software sich ändern. Für den Endbenutzer bedeutet es, dass sich eventuell die Bedienung ändern kann oder das diese Version zu älteren Versionen inkompatibel ist. Für externe Entwickler, welche die Software in eigenen Projekten nutzen, bedeutet es, dass sie ihre eigene Software an die geänderten Schnittstellen anpassen müssen.

`Minor` würde bedeuten, dass sich die Software geändert hat aber in dem Maße, dass eine Änderung von `Major` nicht gerechtfertigt ist. Die Zahl `Revision` wären optional nutzbar. Sie würde verwendet, wenn Fehler oder Sicherheitslücken behoben werden.

Bei sehr jungen Projekten ist es üblich für `Major` den Wert „0“ zu verwenden. Man möchte damit symbolisieren, dass die Software noch nicht stabil ist und das die Schnittstellen der Software noch starken Änderungen unterworfen sind. Denkbar aber unüblich sind auch negative Versionsnummern [Wik11b].

Anhand der Versionsnummern kann man den Status der Veröffentlichung kodieren. So könnte eine gerade Zahl bedeuten, dass es sich um eine stabile Version handelt. Entsprechend könnte eine ungerade Zahl bedeuten, dass diese Veröffentlichung sich nicht für den produktiven Einsatz eignet. Eine weitere Möglichkeit ist, Schlagwörter („alpha“, „beta“, „rc“) der Versionsnummer anzuhängen.

Manche Software-Projekte lassen sich sog. „Snapshots“ beziehungsweise sog. „Nightly Build“ automatisch erstellen. Es bietet sich an hierfür das Datum als Versionsnummer zu kodieren („Wine 20040505“). Als Versionsnummer bietet sich auch das Jahr an wie es bei Windows 95 und Windows 98 beispielsweise der Fall war. Die Linux-Distribution „Ubuntu“ nutzt als erste Zahl das Jahr und als zweite Zahl den Monat („Ubuntu 11.04“).

Oft sind aber Jahreszahlen oder aber auch Buchstabenfolgen, wie am Beispiel von „Microsoft Office XP“ dem Marketing zu verdanken. Man erhofft sich dadurch einen besseren Absatz der Software [Wik11b].

Einen spannenden Ansatz verfolgt TEX. Da die Software als sehr ausgereift gilt, wird bei jeder neuen Version eine weitere Nachkommastelle der Zahl π genommen [Wik11b]. Zum Beispiel die Versionsnummer 3.1415926.

Sinnvoll ist es einen Leitfaden zur Versionierung zu erstellen. Darin soll festgehalten werden nach welchen Kriterien Versionsnummern vergeben werden. Es soll geklärt werden nach welchem Maßstäben eine Version als stabil oder als nicht stabil deklariert wird. Der Leitfaden kann klären welche Abstufungen es geben soll. Eine einfache und zeitsparende Möglichkeit würde darin bestehen, den Leitfaden in Form einer Textdatei in dem Repository der Versionsverwaltung mit zu verwalten.

3.4 „Waf“ als Build-System

Waf ist ein auf Python basierendes Framework zum Konfigurieren, Übersetzen und Installieren von Softwareprojekten [Goo11]. Es erkennt automatisch, in welcher Reihen-

folge Quelltexte übersetzt werden müssen. Es führt selbständig Aufgaben parallel aus. Es hat von Haus aus viel Unterstützung für unterschiedlichste Compiler und Sprachen. Jedoch ist es auch mit der Programmiersprache Python einfach erweiterbar. Das Projekt ist zwar noch sehr jung aber schon zum jetzigen Zeitpunkt gut dokumentiert. So existiert auf der Projektseite bereits ein Link zu einem sehr ausführliches „The Waf book“ [Tho10].

Die Basis stellt der Befehl `waf` dar. Es gibt vordefinierte Phasen die in einer Datei `wscript` konfigurierbar sind. Die wichtigsten Phasen habe ich aufgelistet und mit einer kurzen Beschreibung versehen.

- `configure`: Sucht nach Bibliotheken, Compiler, etc.
- `build`: Übersetzt das Projekt
- `dist`: Erstellt ein Archiv mit dem gesamten Quelltext
- `clean`: Entfernt alle mit `build` erzeugten Dateien

Nun möchte ich an einem einfachen Beispiel aufzeigen, wie Waf in der Praxis nutzbar ist. Im Listing 2 sehen wir ein C-Quelltext, dass „Hello World“ ausgeben soll.

Listing 2: main.c

```
1 #include <stdio.h>
2 int main (void)
3 {
4         puts ("Hello World!");
5         return 0;
6 }
```

Im Listing 3 steht ein Beispiel um den C-Quelltext zu übersetzen. In der Zeile 1 und 2 definieren wir als erstes den Namen des Projektes und geben ihm eine Versionsnummer. In Zeile 3 haben wir eine Funktion `option` mit einem Parameter, dem Optionskontext, definiert. Diese Funktion wird einmalig aufgerufen, um den Compiler für das C-Programm zu suchen. In Zeile 5 bekommen wir mit `cnf` einen Konfigurationskontext. Anschließend werden die Parameter zusammengestellt, die notwendig sind um unser Programm zu übersetzen. Schließlich geben wir in Zeile 7 an, dass wir ein C-Programm übersetzen wollen. Wir geben den Dateinamen von unserem Quelltext an und wie das Programm später heißen soll.

Wie wir sehen, bekommen wir in jeder Phase einen Kontext als Parameter übergeben. Dieser Kontext-Parameter bietet uns die Möglichkeit zwischen den Phasen Informationen auszutauschen.

Im Listing 4 möchte ich nun Schritt für Schritt die Nutzung von Waf beschreiben. In der ersten Zeile führen wir in der Shell den Befehl `waf configure build` aus. Nun sucht Waf zunächst den passenden Übersetzer, in unserem Falle GNU Compiler Collection (GCC). Damit hat er die Konfiguration abgeschlossen und führt im zweiten Schritt

Listing 3: wscript

```
1 APPNAME = 'wafhello'
2 VERSION = '1.0'
3 def options(opt):
4         opt.load('compiler_c')
5 def configure(cnf):
6         cnf.load('compiler_c')
7 def build(bld):
8         bld(features='c cprogram', source='main.c', \
9             target='hello')
```

das Übersetzten des Quelltextes aus. Nach diesem Schritt haben wir nun das Programm `hello` im Verzeichnis `build/` erstellt. In Zeile 11 führen wir das Programm aus und bekommen wie erwartet die Ausgabe „Hello World!“. Mit einer Eingabe `waf clean` in der Shell würden wir die eben erstellten Dateien löschen.

In Zeile 13 führen wir auf der Shell den Befehl `waf dist` aus und wollen damit ein Archiv erzeugen, welches wir beispielsweise per E-Mail versenden können oder auf einer Website veröffentlichen können. Die Ausgabe signalisiert uns, dass eine Datei mit dem Name `wafhello-1.0.tar.bz2` erzeugt wurde. Waf gibt zusätzlich noch einen nützlichen Sha-Hashwert von der Datei aus.

Es fällt auf, dass Waf nach dem Ende einer Phase eine Zeitangabe macht, wie lange er für diesen Schritt gebraucht hat.

Listing 4: Beispiel-Projekt in einer Shell mit Waf übersetzen

```
1  max@muster ~/repo$ waf configure build
2  Setting top to                           : .
3  Setting out to                           : ./build
4  Checking for 'gcc' (c compiler)          : ok
5  'configure' finished successfully (0.026s)
6  Waf: Entering directory `./build'
7  [1/2] c: main.c -> build/main.c.0.o
8  [2/2] cprogram: build/main.c.0.o -> build/hello
9  Waf: Leaving directory `./build'
10 'build' finished successfully (0.061s)
11 max@muster ~/repo$ ./build/hello
12 Hello World!
13 max@muster ~/repo$ waf dist
14 New archive created: wafhello-1.0.tar.bz2
15 (sha='10d691febb5f87e32f3f599749be12df737158a7')
16 'dist' finished successfully (0.012s)
```

3.5 Kleine Projekte

In kleinen Projekten stellt sich oft die Frage, wie groß man den Aufwand betreiben möchte. Jedoch kann auch ein Ein-Mann Projekt von den in diesem Kapitel vorgeschlagenen Vorgehensmethoden nutzen ziehen. Am Anfang steht eine Versionsverwaltung. Es sollte die erste Tat sein, diese einzurichten.

Eine Dokumentation lässt sich mit einer Versionsverwaltung in Form von einfache Textdateien realisieren. In einer Datei README.txt oder Changelog.txt lässt sich wunderbar den Fortschritt der Software dokumentieren. Einen Leitfaden zum Release der Software kann man nach und nach ausbauen und den Anforderungen anpassen. Durch die Versionierung lassen sich Änderungen im Release-Management problemlos nachvollziehen.

Je nach Größe lassen sich Werkzeuge wie Makefile, Apache Ant, GNU Build System oder das eben vorgestellte Build-System Waf einsetzen. Diese Werkzeuge sind freie Software und lassen sich den eigenen Anforderungen anpassen. Eine Reihe von Tutorials ermöglichen einen einfachen Einstieg.

Der Release-Manager spielt in kleinen Projekten oftmals eine andere Rolle als in großen Projekten obwohl sie die gleichen Verantwortung tragen [MHP07]. Während in großen Projekten der Release-Manager Aufgaben delegieren kann und mehr für die Koordination zuständig ist, hat man solche Möglichkeit bei einem kleinen Projekt nicht. Dadurch fällt bei einem kleinen Projekt ein gewisses Maß an Mehrarbeit weg.

4 Beispiele aus der Freien-Software-Szene

In diesem Kapitel möchte ich drei Projekte aus der freien Software-Szene beleuchten. Mittlerweile gibt es eine ganze Reihe von Projekten in der freien Software-Szene, welche ein etabliertes Projekt- und Release-Management besitzen. Ich habe mich hier für den Linux Kernel, Subversion und den Apache HTTP Server entschieden. Diese drei Projekte sind sehr bekannt. Anhand der Verbreitung und den bisher veröffentlichen Versionen ist anzunehmen, dass sie ein erfolgreiches Release-Management haben.

4.1 Linux Kernel

Der Linux Kernel ist der wichtigste Bestandteil eines GNU/Linux Systems. Die ersten Zeilen Quelltext wurden 1991 von Linus Torvalds geschrieben. Linus Torvalds ist noch heute hauptverantwortlich für das Projekt. Seit 1991 ist eine große Gemeinschaft um das Projekt gewachsen. Viele Freiwillige und viele abgestellte Mitarbeiter von IT-Firmen entwickeln an dem Projekt mit. Der Quelltext ist unter der „GNU General Public License“ lizenziert. Die Entwickler des Projektes bezeichnet ihre Software als Klon des Unix-Betriebssystems [ker11].

Die Versionierung ist folgendermaßen aufgebaut: `2.Major.Minor[.Revision]`. Die „2“ in der Versionsnummer ist historisch bedingt und sagt aus, dass die Software als ausgereift gilt. Die Zahl `Major` wird verändert, wenn Änderung an der Systemarchitektur gemacht werden. Es gilt dabei, dass die Veröffentlichungen mit geraden Zahlen (zum Beispiel 2.4 oder 2.6) für den produktiven Einsatz gedacht sind. Veröffentlichungen mit

ungeraden Zahlen sind Versionen, welche ausschließlich zum Experimentieren bestimmt sind. So war die Vorgängerversion der 2.6. Kernel-Serie mit der Versionsnummer 2.5 deklariert. Ob es eine Kernel-Version mit der Version 2.7 oder 2.8 geben wird, ist zu diesem Zeitpunkt fraglich. `Minor` wird inkrementiert, wenn neue Funktionen implementiert werden. `Revision` ist optional und wir nur dann verwendet, wenn nach der Freigabe Fehler behoben werden müssen.

Auf der Website vom Projekt ist ersichtlich welche Version welchen Status erreicht hat. Es sagt dem Endbenutzer aus in welchem Zeitraum er Unterstützung in Form von Fehlerbehebung der Entwickler erfahren kann. Linux-Distributoren können also anhand vom Status entscheiden, welche Version für ihre nächste Veröffentlichung interessant ist. So gibt es folgende Bezeichnungen:

- `linux-next`: Testet mögliche zukünftige Implementierungen
- `mainline`: Hauptzweig unter der Verantwortung von Linus Torvalds
- `stable`: Veröffentlichte Version die ausschließlich gepflegt wird
- `longterm`: Veröffentlichte Version die über einen längeren Zeitraum gepflegt wird.

Die Verantwortung für den Hauptzweig liegt nach wie vor bei Linus Torvalds. Ist die Freigabe einer neuen Version erfolgt, wird die Verantwortung an Entwickler abgegeben, die nach Einschätzung von Linus Torvalds, dafür geeignet sind. Diese Entwickler haben dann die Aufgabe die jeweilige Version weiter zu pflegen und mögliche Fehler zu beheben.

In dem `linux-next`-Zweig werden die neuesten Funktionen integriert und getestet. Auch dieser Zweig wird von einem erfahrenen und vertrauten Entwickler gepflegt. Linus Torvalds übernimmt diese Funktionen in den Hauptzweig, wenn er der Meinung ist, dass diese ausreichend getestet wurden. In der Regel werden nach zwei Wochen keine neuen Funktionen in den Hauptzwei übernommen. Es findet also ein Einfrieren des Quelltextes statt. Nun wird das Hauptaugenmerk auf das Test und Härten der Software gelegt. Über eine Mailingliste wird evaluiert inwiefern diese neuen Funktionen brauchbar sind. Die Abnahme erfolgt schlussendlich durch Linus Torvalds selbst, der darüber entscheidet ob eine Freigabe sinnvoll ist. Bei älteren Versionen, die ausschließlich gepflegt werden, wird analog vorgegangen.

Eine Philosophie der Entwickler ist möglichst früh und möglichst oft eine Veröffentlichung vom Linux Kernel vorzunehmen [Ray99]. Zu diesem Zweck wurde von den Entwicklern das Projekt Git ins Leben gerufen. Der gesamte Quelltext wird mit dieser Versionsverwaltung verwaltet. Viele Entwickler haben unter `http://git.kernel.org/` ein Repository und experimentieren mit dem Quelltext vom Linux Kernel. Es ist offensichtlich, das die Versionsverwaltung ein ganz wesentlicher Bestandteil des Release-Managements ist.

Wenn die Software freigegeben wird, erfolgt eine Ankündigung auf der Mailingliste. Die Archive mit dem Quelltext und der letzte Commit in der Versionsverwaltung werden vom Release-Manager mit GnuPG signiert. Hierdurch wird die Integrität gewährleistet. Auf zahlreichen Spiegelserver und auf der Website vom Projekt werden Archiv-Dateien (tar.bz2) und Patches angeboten.

4.2 Subversion

Subversion ist eine Software zur Versionsverwaltung. Die erste Version wurde im Jahr 2000 veröffentlicht. Das Projekt wurde begonnen, da man unzufrieden mit dem Vorgänger Concurrent Version System war. Es wird von CollabNet Inc. weiter entwickelt und wird seit 2009 von der Apache-Software-Foundation unterstützt [The09]. Es ist unter der „Apache License v2.0“ lizenziert. Das Projekt hat einen Leitfaden in dem beschrieben ist, nach welchen Regeln die Weiterentwicklung erfolgen soll und wie das Release-Management auszusehen hat.

Das Projekt ist so organisiert, das es drei verschiedene Rollen innerhalb des Projektes gibt [The11a]. Einmal gibt es die Committers, die das Recht haben neue Implementationen in die Software einzupflegen. Dabei werden zwischen zwei Arten von Committers unterschieden. Die einen mit vollem Schreibzugriff auf den Quelltext und solche, die nur partiell Schreibrechte erhalten. So gibt es Committers die nur ein bestimmten Module entwickeln. Desweiteren gibt es einen „Patch Manager“. Er hat die Aufgabe die Mailingliste zu beobachten und sämtliche Mails, die Patches enthalten, zu lesen und zu bewerten. Schlussendlich gibt es noch den Release-Manager. Er hat die Aufgabe die Veröffentlichungen zu koordinieren. Dabei stehen das Härten der Software beziehungsweise das Überprüfen der Checklisten im Vordergrund.

Auf der Website von Subversion sind viele Vorgänge dokumentiert. So gibt es ein Dokument über die verschiedenen Rollen, ein Dokument über das Übersetzten und Testen der Software sowie ein Dokument wie ein Release ablaufen sollte. Im Repository befinden sich weitere nützliche Dokumente zum Release-Management. Beispielsweise ist ein Leitfaden zu finde, wie ein Changelog am Besten geschrieben werden kann.

Die Versionsnummer ist nach dem Prinzip `Major.Minor.Patch` aufgebaut und besteht ausschließlich aus Ganzzahlen. Die `Major`-Versionen sind nicht kompatibel untereinander. Sie wird inkrementiert wenn es wesentliche Änderungen der Schnittstellen gibt. `Minor` wird bei jeder neuen Veröffentlichung inkrementiert. Die Schnittstellen bei einem `Minor`-Release bleiben untereinander kompatibel. `Patch` steht bei der ersten Freigabe auf dem Wert „0“. Sobald nach der Freigabe Fehler entdeckt und behoben werden wird diese Zahl inkrementiert.

Alle Versionen werden grundsätzlich als stabil bezeichnet. Vor der Freigabe gibt es jedoch eine Release-Kandidat-Phase die zum Testen und zum Härten gedacht ist. Falls es der Release-Manager für sinnvoll hält, kann es eine Alpha- und eine Beta-Phase geben, die nicht stabil ist. Der Sinn von solchen Phasen ist, dass neue Funktionen von der Gemeinschaft ausführlich getestet und bewertet werden um dann anschließend darüber zu Entscheiden ob die neuen Funktionen beibehalten oder entfernt werden sollen.

Sobald ein Release-Kandidat veröffentlicht wurde, beginnt eine Zeitspanne von vier Wochen. In dieser Zeit werden regelmäßig neue Vorabversionen zum Testen bereit gestellt. Sobald kritische Fehler entdeckt werden oder anderweitig Änderungen vorgenommen werden, welche die Stabilität der Software gefährden, wird diese Zeitspanne verlängert. Besteht Uneinigkeit darüber ob eine Verlängerung der Testphase notwendig ist, befindet die Gruppe der Committers per Wahl darüber.

Wie eben erwähnt findet bei Uneinigkeit, aber auch bei tiefgreifenden Änderungen

am Quelltext eine Wahl innerhalb der Entwicklergemeinschaft statt. Dazu schicken die einzelnen Committers eine Mail an die Mailingliste. Eine positive Entscheidung ist getroffen, sobald drei positive (+1) Stimmen und kein Veto (-1) eingetroffen sind [The11a]. Die betroffene Version wird als nicht stabil deklariert und die Testphase beginnt von neuem.

Für jeden gefunden Fehler soll ein Test geschrieben werden der diesen Fehler reproduziert. Der Test wird in das Repository eingepflegt und von einer Build-Farm laufend überprüft. Im Repository gibt es einen Leitfaden wie Tests auszusehen haben. Für die Abnahme wurde als Kriterium festgelegt, dass sämtliche Punkte im Issue-Tracking-System als erledigt markiert sein müssen und das sämtliche Tests erfolgreich ablaufen müssen. Der Quelltext der Software wird dabei auf unterschiedlichen Plattformen und System übersetzt. Erst wenn diese Kriterien erfüllt sind, wird der Release-Kandidat als stabile Version freigegeben.

Auf den Servern von collabNet Inc. wird der Quelltext der Software zum Herunterladen angeboten. Dabei werden die Archiv-Formate Tarball-Bunzip und Zip verwendet. Die Firma collabNet Inc. bietet keine binäre Pakete an, sonder verweist auf dritte Anbieter die solche Pakete bereitstellen.

4.3 Apache HTTP Server

Der Apache HTTP Server ist einer der populärsten Webserver. Der erste Quelltext entstand 1995. Seit dem hat sich eine große Gemeinschaft gebildet, die das Projekt weiterentwickeln. Die Software ist unter der „Apache License v2.0“ lizenziert.

Das Projekt ist meritokratisch organisiert. Das heißt, dass man in die Entwicklergemeinschaft aufgenommen wird aufgrund von Leistungen. Es gilt also das Prinzip, je mehr Arbeit man geleistet hat, desto mehr Rechte bekommt man innerhalb des Projektes [The11b]. Wie auch bei dem Projekt von Subversion existiert ein Leitfaden wie das Release-Management vonstatten gehen soll.

Es gibt zwei unterschiedliche Gremien. Den Projekt-Management-Ausschuss und eine Committers-Gruppe. Um Teil des Projekt-Management-Ausschusses zu werden, benötigt es den Konsens aller Mitglieder innerhalb dieses Ausschusses. Um in die Gruppe der Committers zu kommen, benötigt es ebenso den Konsens der Mitglieder im Projekt-Management-Ausschusses. Aus sämtlichen Mitgliedern des Projekts muss sich ein Freiwilliger finden, der ein Releaseprozess startet. Letztlich entscheiden die Projekt-Manager ob ein Release sinnvoll ist oder ob eventuell ein späterer Zeitpunkt besser ist.

Das Projekt unterscheidet zwischen drei Klassifizierungen [The11c]:

- **Alpha:**
  Version ist nicht für den Endbenutzer gedacht
- **Beta:**
  Version steht kurz vor der Fertigstellung und wird demnächst als „General Availability“ veröffentlicht

- **General Availability (GA):**
  Beste bisherige Version und kann in produktiven Umgebungen eingesetzt werden.

Stabile Versionen werden mit `Major.GeradeZahl.Revision` gekennzeichnet. `Major` hat zur Zeit die Zahl „2“ und wird nur bei größeren Änderungen erhöht. Falls eine Erweiterung der Software als Experimentell eingestuft wird aber die Version an sich als stabil angesehen werden kann, bekommt die Versionsnummer den Zusatz „GA“. Eine stabile Version garantiert auch, dass die Kompatibilität zu späteren höheren Versionen gewahrt bleibt. `Revision` wird inkrementiert bei späteren Bereinigungen von Fehlern und steht zu beginn auf dem Wert „0“. Entsprechend bekommen nicht stabile Versionen ein `Major.UngeradeZahl.Revision` als Versionsnummer. Zusätzlich wird „alpha“ oder „beta“ angehängt um den Stand der Veröffentlichung zu kennzeichnen.

Ein Teilprojekt vom Apache HTTP Server beschäftigt sich ausschließlich mit dem Thema Testen. Es konzentriert sich darauf Test-Werkzeuge für das Projekt zu entwickeln [The11d]. Bevor eine Freigabe erfolgen kann, sollten die Tests von diesem Teilprojekt erfolgreich bewältigt werden.

Hat die Software eine Stabilität erreicht die über einen Beta-Status hinausgeht, empfiehlt der Leitfaden zum Release-Management einen letzten Test. Die Website der Apache-Software-Foundation (`apache.org`) soll für 48 bis 78 Stunden mit dem Release-Kandidaten betrieben werden [The11c].

Die Freigabe einer neuen Version erfolgt durch eine Wahl der Mitglieder der Entwicklergemeinschaft. Es müssen mehr positive Stimmen als negative Stimmen gezählt werden und es müssen mindestens drei Mitglieder ein positives Votum abgeben.

Es werden ein Archiv mit dem Quelltext und binäre Pakete veröffentlicht. Zusätzlich werden diese vom Release-Manager signiert. Nach einer Zeitspanne von 24 bis 48 Stunden wird die Veröffentlichung auf der Mailingliste der Entwickler und auf einer für Ankündigungen eingerichteten Mailingliste angekündigt. Die Zeitspanne soll dazu dienen, dass Spiegelserver die Version schon kopiert haben und so Überlastung der Server vorgebeugt wird.

Anders als beim Linux Kernel und bei Subversion, werden hier binäre Pakete für diverse Plattformen angeboten. Jedoch sind sie nicht Teil des Kriteriums. Das heißt, dass in jedem Fall die Ankündigung der neuen Version gemacht wird auch wenn das eine oder andere binäre Paket noch nicht zur Verfügung steht.

Bemerkenswert sind die vielen Spiegelserver weltweit. Das eigene Serverangebot dient dazu diese Spiegelserver zu versorgen. Möchte man eine Version von der Projektseite herunterladen, wird man immer auf einen Spiegelserver verwiesen.

4.4 Vergleich der drei Software-Projekte

Zum Abschluss von diesem Kapitel möchte ich das Release-Management von den drei vorgestellten Projekten vergleichen. Auffallend ist, dass alle drei Projekte eine ganz unterschiedliche Herangehensweise an das Thema haben. Dies ist sicher auch dadurch bedingt, dass alle drei Projekte einen unterschiedlichen historischen Hintergrund haben.

Der Linux Kernel ist sehr dezentral organisiert. Es ist sehr wenig Dokumentation vorhanden zum Thema Release-Management. Im Mittelpunkt steht die Versionsverwaltung Git und die Mailingliste der Entwickler. Viele Abläufe sind über die Jahre gewachsen und haben sich als praktikabel erwiesen. Der Gründer, Linus Torvalds, steht nach wie vor im Mittelpunkt. Sein Wort hat sehr viel Gewicht innerhalb der Entwicklergemeinschaft. Er bestimmt die Richtung und das Vorgehen, auch beim Release-Management.

Subversion hingegen ist sehr zentral organisiert. Obwohl die Apache-Software-Foundation einen Großteil der Infrastruktur liefert, ist die Gründerfirma CollabNet Inc. nach wie vor stark in die Entwicklung eingebunden. Das Projekt besitzt eine gute Dokumentation die klar das Release-Management beschreibt. Zu jeder Fehlerbeschreibung muss ein Testskript folgen, welches automatisiert überprüft wird. Dazu wird eine Build-Farm verwendet welche ständig die Software auf unterschiedlichen Plattformen übersetzt und einen Regressionstest durchführt. Die Gemeinschaft kommuniziert über Mailinglisten und nutzt diese für Wahlen, wenn wichtige Entscheidungen zum Projekt anstehen. Es ist anzumerken, dass das Projekt stark von der Infrastruktur der Apache-Software-Foundation profitiert.

Der Apache HTTP Server ist nicht so zentral organisiert wie Subversion. Ein Ausschuss von Projekt-Managern lenkt das Projekt und bestimmt im wesentlichen über das Schicksal des Projektes. Die Entwicklung ist dezentral organisiert und wer sich stark einbringt, wird für seine Arbeit belohnt. Es ist eine gute Dokumentation über Release-Management vorhanden. Sie ist aber freibleibend gestaltet, sodass man bei unvorhersehbaren Schwierigkeiten als Release-Manager nicht zwingend an die Projekt-interne „Demokratie“ gebunden ist.

Das Projekt betreibt, im Vergleich zu den beiden anderen Projekten, den meisten Aufwand um unterschiedliche Plattformen zu bedienen. So werden binäre Pakete für unterschiedlichste Plattformen angeboten. Der Linux Kernel und Subversion bieten ausschließlich den Quelltext an. Immerhin verweist Subversion auf dritte Anbieter die binäre Pakete anbieten.

Subversion und der Apache HTTP Server haben in ihrer Dokumentation Wahlen definiert. Jeder Entwickler kann dadurch Einfluss auf den Release-Kandidaten nehmen. Jedoch besteht auch die Gefahr, dass sich Entwickler gegenseitig blockieren und dass es zu keiner Veröffentlichung neuer Versionen kommt.

5 Bezug zur Softwareentwicklung in der Wissenschaft

In diesem Kapitel möchte ich Release-Management mit dem Thema „Softwareentwicklung in der Wissenschaft“ verbinden. Wissenschaftler haben häufig keine oder eine geringe Ausbildung im Bereich der Softwareentwicklung [EJ09]. Im Gegenzug müssen Softwareentwickler sich meistens erst in das jeweilige Forschungsfeld einarbeiten. Softwareentwickler benötigen ein tiefgehendes Verständnis für das Problem welches untersucht werden soll. Das bedingt eine gute Kommunikation zwischen Wissenschaftler und Softwareentwickler. Freie Software-Projekte können da als gutes Vorbild dienen. Die, in dieser Seminararbeit, besprochene Projekte in Kapitel 4 haben ein über Jahre etabliertes Release-

Management. Ein sinnvoller Einsatz von Release-Management kann Nachhaltigkeit in der Wissenschaft sichern.

Um den Nutzen von Release-Management besser verstehen zu können, ist es notwendig ein wenig auf das Vorgehen in der Softwareentwicklung in der Wissenschaft zu beleuchten. In der Wissenschaft werden theoretische Modelle kontinuierlich weiterentwickelt und es wird versucht diese in Software umzusetzen. Hierdurch wird der Versuch unternommen Hypothesen zu untersuchen [EJ09]. Das heißt, dass Wissenschaftler eine kontinuierliche Weiterentwicklung ihrer Software haben und Testabläufe dafür nutzen um ihre Modelle zu verifizieren. Wir haben es mit Experten in einer bestimmten Domäne zu tun. Sie sind nicht als Programmierer eingestellt worden, aber müssen sich mit Programmierung beschäftigen. Häufig hat nur ein geringer Personenkreis vollen Zugriff auf den Quelltext [EJ09]. Das hat die Gefahr das der Truckfaktor zuschlägt — bei einem Ausfall einer Person steigt die Wahrscheinlichkeit stark, dass das Projekt scheitert. Bei der Softwareentwicklung in der Wissenschaft ist der zeitliche Aspekt nicht so kritisch zu betrachten als bei einer Softwareentwicklung im kommerziellen Kontext.

Die Pflege der Dokumentation wird häufig vernachlässigt und Wissen wird oft auf dem „kurzen Dienstweg“ ausgetauscht [EJ09]. Es ist wichtig die eigene Dokumentation aktuell zu halten. Leitfäden und Checklisten bieten eine gute Möglichkeit nerven- und zeitraubende Fehler zu vermeiden. Auch hier bieten freie Software-Projekte eine gutes Beispiel. Die Dokumentation in den oben genannten freien Software-Projekten kann als Anleitung dienen, um ein eigenes Release-Management zu etablieren.

Durch Release-Management ist es möglich die Reproduzierbarkeit von Ergebnisse zu gewähren. Versionsverwaltung bietet die Möglichkeit, auf ältere Veröffentlichungen der Software zurück zugreifen. So kann man zu jedem Zeitpunkt seine Ergebnisse mit verschiedenen Softwareversionen vergleichen und überprüfen.

6 Zusammenfassung

Release-Management muss geplant und dokumentiert werden. Dabei sind Fragen zu klären. Fragen sind, was man am System konfigurieren muss um die Software zum Laufen zu bekommen oder was nötig ist um die Software installiert zu bekommen. Es sollte definiert werden wie Software vor der Veröffentlichung gehärtet werden kann. Es sollte also feststehen, wie getestet werden soll.

Ein wichtiger Punkt ist die Frage der Versionierung. Eine Versionsverwaltung ist dabei essentiell. Wiederkehrende Aufgaben lassen sich automatisieren beziehungsweise durch Werkzeuge vereinfachen. Meine Beispiele zeigen auf, dass es nicht viel Aufwand benötigt diese einzurichten und zu nutzen.

Es gibt bekannte Projekte aus der freien Software-Szene, die ein funktionierendes Release-Management anwenden. Sie können als gutes Vorbild für eigene Projekte dienen. Manche dieser Projekte haben ihren Veröffentlichungsprozess auf ihren Websites dokumentiert.

Selbst kleine Projekte und auch Ein-Mann-Teams können Release-Management sinnvoll nutzen und hierdurch Arbeit und Zeit sparen. Es benötigt keine aufwendige Software

hierfür. Textdateien, welche den Release-Prozess beschreiben und in der Versionsverwaltung mit verwaltet werden reichen oftmals schon aus. Man bleibt flexibel genug und hat die Möglichkeit auf notwendige Anpassungen im Verlauf der Entwicklung der Software zu reagieren. Wenn einmal der Aufwand betrieben wurde und das Release-Management dokumentiert wurde, lässt sich das oftmals für neue Projekte übernehmen und anpassen.

Es gibt nicht einen optimalen Weg. Jedoch gibt es eine Menge Vorgehensweisen und gute Beispiele. Wenn man ein Release-Management etablieren möchte, lohnt es sich ein wenig zu recherchieren. Viele vorhandene Leitfäden in freien Softwareprojekte lassen sich übernehmen und auf die eigenen Bedürfnisse anpassen. Langfristig lohnt sich diese Arbeit, auch wenn der Nutzen auf den ersten Blick sehr gering erscheint. Diese Aussage ist eine Schlussfolgerung aus meinen Beobachtungen der freien Softwareprojekten.

Abschließend möchte ich sagen, dass man sich in jedem Fall mit Release-Management beschäftigen sollte. In jedem Softwareprojekt sollte man sich Gedanken darüber machen und klären, auf welche Weise man den meisten Mehrwert bekommt.

Literatur

[CSK02] Clement-Smith, L. ; Kerkhoff, W.: Software Release Management. (2002)

[EJ09] Easterbrook, S.M. ; Johns, T.C.: Engineering the software for understanding climate change. In: *Computing in Science & Engineering* 11 (2009), Nr. 6, S. 65–74. – ISSN 1521–9615

[Ere03] Erenkrantz, J.R.: Release management within open source projects. In: *Proc. 3rd. Workshop on Open Source Software Engineering* Citeseer, 2003

[Goo11] Google Project Hosting: *Waf*. `http://code.google.com/p/waf/`, Februar 2011

[Her08] Herbert Braun: *heise online - Download Day*. `http://heise.de/-214976`, 18. Juni 2008

[HW03] Hoek, A. Van d. ; Wolf, A.L.: Software release management for component-based software. In: *Software: Practice and Experience* 33 (2003), Nr. 1, S. 77–98. – ISSN 1097–024X

[ISO04] ISO, DIN: 10007. In: *Qualitätsmanagement– Leitfaden für Konfigurationsmanagement (ISO 10007: 2003)* (2004)

[ISO05] ISO, DIN: 20000-1. In: *Service Management: Specification (ISO/IEC 20000 Part 1: 2005)* (2005)

[ker11] *The Linux Kernel Archives*. `http://www.kernel.org/`, Februar 2011

[MHP07] Michlmayr, M. ; Hunt, F. ; Probert, D.: Release management in free software projects: Practices and problems. In: *Open Source Development, Adoption and Innovation* (2007), S. 295–300

[Ray99] Raymond, E.: The cathedral and the bazaar. In: *Knowledge, Technology & Policy* 12 (1999), Nr. 3, S. 23–49. – ISSN 0897–1986

[The09] The Apache Software Foundation: *Subversion Submitted to Become a Project at The Apache Software Foundation*. `http://www.apache.org/foundation/press/pr_2009_11_04.html`, 4. November 2009

[The11a] The Apache Software Foundation: *Apache Subversion - Community Guide*. `http://subversion.apache.org/docs/community-guide/community-guide.html`, Februar 2011

[The11b] The Apache Software Foundation, Documentation Group: *About the Apache HTTP Server Project*. `http://httpd.apache.org/ABOUT_APACHE.html`, Februar 2011

[The11c] The Apache Software Foundation, Documentation Group: *Apache HTTP Server Release Guidelines*. http://httpd.apache.org/dev/release.html, Februar 2011

[The11d] The Apache Software Foundation, Documentation Group: *Apache HTTP Test Project*. http://httpd.apache.org/test/, Februar 2011

[Tho10] Thomas Nagy: *The Waf Book*. http://waf.googlecode.com/svn/docs/wafbook/single.html, 24. November 2010

[Wie11] Wiesner, Robert: *Versionsverwaltung*. http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2010_2011/siw-2010-wiesner-versionsverwaltung-ausarbeitung.pdf, 13. März 2011

[Wik11a] Wikimedia: *Release Management*. http://de.wikipedia.org/w/index.php?title=Release_Management&oldid=83836978, 1. Februar 2011

[Wik11b] Wikimedia: *Software versioning*. http://en.wikipedia.org/w/index.php?title=Software_versioning&oldid=409736919, 6. Februar 2011

[Wik11c] Wikimedia: *Softwareverteilung*. http://de.wikipedia.org/w/index.php?title=Softwareverteilung&oldid=85097248, 10. Februar 2011