Building Packages

with
Git-Buildpackage

Dipl.-Ing. Mechtilde Stehmann

and

Dr. Michael Stehmann

January 28, 2023

Content

Part I

Overview

Chapter 1

License

Chapter 2

Who should read this book?

The information in this book is of particular interest to users of the script. But also people who are generally interested in packaging for a distribution will find information in this book.

This book does not want to be a "textbook" for building `Debian` packages. It is more of an experience report, where the experiences have been "molded into code".

The book describes how `Debian` packages are created using a git repository with the programs from the package *git-buildpackage* [3] and other useful commands. Afterwards, the reader should be able to use the program as shown and the descriptions of the individual steps to build "production quality" `Debian` packages.

The program script itself does **not** build `Debian` packages, but assists the user in building them. It is only an assistance program.

This book can also be used to understand problems that can arise when packaging `Debian` packages.

Packaging is basically not difficult, though there are always new challenges. Packaging is therefore fun.

Chapter 3

How is this book conceived?

3.1 Motivation

What is driving us to write such a book?

To do this, you need to know the following::

Packaging involves executing many commands in a meaningful order at the shell. In addition, many small files must be maintained and included. The smallest errors and inaccuracies usually result in the package not being built correctly. It's also time-consuming and error-prone to keep putting in the correct options.".

To keep these sources of error as small as possible, these steps were combined in a shell script. In the course of the time and with each further package this script grows now and is refined also always further. So far, this has already become an extensive program script.

When Mechtilde started to deal with building `Debian` packages, the question was how to document and automate these many steps. The need for documentation could not be met from the beginning with comments, neither in the individual files nor in this program script.

That's why we started early on to record our packaging steps in detail. We paid special attention to descriptions that make decisions traceable and verifiable. This makes it easier to make necessary changes..

For this reason, we have also chosen the long form as far as possible when specifying options. This facilitates readability.

So the documentation should describe the actual packaging as well as explain the script.

The `Debian` distribution is the work of many people. It consists of tens of thousands of packages. Building the packages is a major task of the package maintainers. Many package maintainers use their own scripts for this purpose. Publishing such a script is therefore a gamble. If our script makes life easier for some package maintainers and introduces newcomers to package building, this venture has been worthwhile.

The described program script does not refer to a specific `Debian` package. Rather, it is intended to be used to build simple `Debian` packages in general.

It describes the steps we need to take to package the packages maintained by Mechtilde. The program script does not claim that it can be used to build a Debian package from any source code.

In many places, you can and must also intervene manually. The description of the processes during packaging should be of help here.

The fact that the program script presupposes the possibility of manual intervention makes it necessary for the program script to repeatedly check whether the prerequisites that the authors have assumed are actually present. This necessity unfortunately increases the size and complexity of the script and thus also the size of the book.

3.2 Under contruction

The book and the script are still ”under development”, because new experiences keep flowing in.

The book is written in German, the program script interface in English. Localizations are welcome. For an English translation the required **.po* files have already been created..

The release of the source code is done in the `Git`-Repositorium[1] provided by the `Debian` project. There, the *CI/CD* (chapter **??**, page **??**) is enabled. Therefore the finished documents are available there as **.pdf* und **.epub*.

3.3 Tools

“Living more comfortably with documentation” is a common phrase in our *peer group*.

Which tools can be used to create such documentation? Are these tools also available as `Debian` packages?

Mechtilde received an important hint about this at an event for the *Software Freedom Day 2012* in Cologne. There she learned about the possibilities of *noweb* [4][2] This also meant getting closer to LaTeX .

In this combination, it is possible to maintain code and its description in *one* document. Donald Knuth refers to this as "Literate Programming” [3]

Further we have dealt with the fact that LaTeX can be used to create documents in `EPUB` format in addition to PDF documents . These can also be read with an e-book reader .(chapter **??**, page **??**)

Translating this book into another language is a special challenge. Our tests have shown that `OmegaT` [4] is a useful and convenient tool for that purpose. The corresponding process is also documented in this book.

The bibliography is created and maintained with *jabref*. The file created in this way can be included in the LaTeX document.

[1] `https://salsa.debian.org/ddp-team/dpb`

[2] s.a. `https://en.wikipedia.org/wiki/Noweb`

[3] `http://www.literateprogramming.com/`

[4] `https://packages.debian.org/sid/omegat`

The used editor is *geany* with the plugin *geany-plugin-latex*t.

Git is ingenious. Building is therefore done with the tools from the *git-buildpackage*[5] package.

The people at `Debian` have created many useful programs that make building `Debian` packages easier and more uniform. The program script presented was therefore created “on the shoulders of giants”.

It is used to call up the auxiliary programs used in an expedient order and to provide them with useful options. It should show its users the way and make their work easier. To facilitate its adaptation to the needs of its users, it is a shell script.

[5] `https://packages.debian.org/sid/git-buildpackage`

Chapter 4

Conventions

Some notes for a better understanding of the book:

4.1 System

The book and especially the program script were created on a *64-Bit-PC* architecture. This is called *amd64* in `Debian`. Another name for this system is *x86-64*.

4.2 Terminology

A **new package** is a package, which the program script does not know yet. I.e. to this package no configuration file exists yet.

A **new version** is a new upstream version. Building a new version is followed by building a new revision.

A **new revision** denotes a new `Debian`-package to be uploaded.

4.3 Typography

All program names are set in *italic*.All proper names are set in `nonproportional` type. Superscript numbers point to footnotes on the same page. Citations to an entire document point directly to the bibliography in square brackets [ ].

All shell command options are given in the long form, as far as this is possible. This increases the readability.

For the abbreviations used, please refer to the entries in the glossary[1].

4.4 Source Code Representation

The source code is presented in segments (so-called *code chunks*). The order of these code chunks in the book often does not correspond to the order in

[1] `https://wiki.debian.org/Glossary`

the scripts. The fact that the order in the book and script need not match is a merit of *Noweb*.

Part II

Preparation

This part of the book starts with a little theory. Then the setup of the system including the Git repository, which is essential for *git-buildpackages*, is described.

Chapter 5

Literature

Some people ask what they should have to read before starting building Debian packages. Others want to know where they can find helpful information. So here are recommended readings for those who (want to) build Debian packages.

A short introduction to "packaging for Debian" is given in the "Simple Packaging Tutorial"[5].

The following three documents are "must-reads" for any package maintainer.

5.1 Debian Free Software Guidelines

All who want to contribute to Debian are recommended to read the social contract. The *Debian Free Software Guidelines (DFSG)*[6] are an integral part of the social contract. They contain the conditions that a license must meet to be considered "free" They are already significant in the selection of the software to be packaged (chapter 8, page 25).

The *Debian Free Software Guidelines (DFSG)* apply not only to software licenses, but according to clause 1 of version 1.1 of the Social Contract "all its components") also to the licenses of images, sounds, texts etc.

5.2 Debian Policy Manual

The Debian Policy Manual[7] describes the policies for the Debian GNU/Linux distribution. It describes the structure and contents of the Debian archive, various operating system design decisions, and technical requirements that each package must meet to be included in the distribution. This document is available in English only.

It is also available as Debian package *debian-policy*.

The most important addition to the Policy Manual is the "Filesystem Hierarchy Standard"[8].

There are also other supplementary sets of rules.[1]

[1] https://www.debian.org/devel/index.en.html

5.3 Debian Developer Reference

The Developer Reference[9] lists the procedures and resources for Debian-developers. The document describes how to become a new developer for the Debian project, the upload procedure, how to operate the bug database (bug tracking system), the mailing lists, Internet servers, etc.

This document is intended as a reference manual for all Debian developers. It is available as Debian package *developers-reference-en*. [9]

5.4 Reference for *Git-Buildpackage*

There are various procedures and tools for building Debian packages. In this book, *git-buildpackage* is used.

In addition, we therefore still recommend the *git-buildpackage*[3] reference for the build system we have chosen.

5.5 Manual for Debian Maintainer

This manual[10] describes how to build a Debian package using the *debmake* command. It is intended for normal users and aspiring package maintainers.

The focus is on the modern packaging style. Many simple examples are included.

This "Manual for Debian maintainers" should consider as a legacy of the "Debian Guide for New Package Maintainers".

It is also available as Debian package *debmake-doc*. [10]

5.6 Debian New Maintianer Guide

This mature work[11] attempts to describe building Debian packages in a way that is understandable to ordinary users and future developers, with working examples.

Unlike previous documents, this one builds on *debhelper* and other tools available to a developer. [11]

This document is also available in other languages and as Debian packages.

5.7 More Information

Other useful literature can be found at https://www.debian.org/doc.

On the subject of building Debian packages, one can find documents in other places on the Internet. However, it is important to note: "The Internet does not forget anything, and somewhere there is always still an

outdated documentation linked, whose obsolescence is also not recognizable due to the lack of an expiration date." [2][12]

[2]About this book, chapter 2.9 in the book Debian Package Management

Chapter 6

What is a Debian Package?

To understand the way, you should know the goal. The goal of packaging is a Debian package[5].

But what is a Debian package?

A formal answer is: A Debian package is a software package released by the Debian project.

For a package to be released by the Debian project, it must meet requirements established, written, and published for transparency by the Debian project.

A Debian package is a software package that primarily satisfies the requirements described in the Debian policy[7]. It is part of the transparency maintained by the Debian project that the exact version of the Debian policy followed in building the package is specified in the *debian/control* file. (Chapter **??**, page **??**)

A Debian packages is a collection of files that allow applications or libraries to be distributed through the Debian package management system. The goal of packaging is to allow the automation of installing, updating, configuring, and removing software for Debian in a consistent manner.[5]

A Debian package is more than just an archive file containing executable code with the extension *.deb*. A Debian package consists of **four** files; three of them are archives.

- An archive file with the extension *.orig.tar.gz* resp. *.orig.tar.xz* contains the source code from which the package is built.
- Another archive file with the extension *.debian.tar.xz* contains the debian-specific files that control the build process and installation or contain additional information.
- A file contains the signatures of the archive files. This file has the extension *.dsc*. This file signed, too.
- Finally, the executable code is in the archive with the extension *.deb*.

The Debian package system enables traceability of the path from the upstream source code to the binary Debian package. Aimed at and often achieved is a bit-accurate reproducibility of the build process[13].

How this can be checked is described in chapter 22 (page 91).

This transparency gives the user confidence that the binary package was also comprehensibly built from the published source code.

If you save software from your build system to an archive in deb format without caring about standards and rules, you don't pack a `Debian` package.

Several variants (releases) are offered by the `Debian` project at any time, namely *stable*, *testing* and *unstable*. After the release of each new *Stable* version, the previous *Stable* version is maintained for some time as *Oldstable* . Furthermore, there is still *oldoldstable* and a branch *experimental*. There, changes are tried out that could have serious effects on the overall system.

However, the *experimental* branch is not a complete distribution, but works only as an extension of *unstable.* [14]

The branch *unstable* (*Sid*) is the first port of call for new programs and new versions of programs before they are integrated into *testing*.

The development of a `Debian` package starts in the *unstable* branch.[1]

Each `Debian` package belongs to a defined development stage of the `Debian` distribution, i.e. to a specific, released version. These are called releases.[2].

Each package must be aligned with this release. It must not have any dependencies on another release. Libraries may only exist in one version in a release, so that security updates are easy for the user. A `Debian` package must therefore not contain its own version of such a library. [3]

[1] I.Concepts, chapter 2.10.1[12]
[2] Chapter 2.10 in [12]
[3] Chapter 7.1 [15]

Chapter 7

How to select the software to be packaged

Often the question is asked to the project, which package is suitable for starting. To learn packaging, there is no specially created "training" package. Rather, one learns packaging by packaging `Debian` packages ("learning by doing").

It is emphasized from the beginning that the motivation is brought along to work step by step into the subject of packaging. A good condition for it is also that it is a software, which one would like to use gladly as `Debian` package and make available to others..

Often there are also packages for which the maintainers need help or which are orphaned. This can also apply to packages that are used by themselves. To find out if and which of the installed packages are affected, there is a tool that you can install additionally as package *how-can-i-help* [16].

This installation causes *apt* to call *how-can-i-help --apt* at the end. This will then list help-needing packages that have just been updated.

With *how-can-i-help --old* you can see which of the installed packages need help.

With *how-can-i-help --all* all packages are displayed for which help is needed[1].

This information can also be found at `https://www.debian.org/devel/wnpp/`.

It is also motivating and helpful to be active in the Upstream project.

In these cases, a certain sustainability of the care of the package is guaranteed.

For many categories of packages support teams have been formed [2]. If you choose a package that fits into the portfolio of such a team, you have a better chance to find support and sponsors. Support can also be found on

[1] More details about *how-can-i-help* are described in Debian Package Management[12] in chapter 37.3.6.

[2] Lists can be found at `https://wiki.debian.org/Teams`.

the mailing lists of the respective teams.

For all questions about packaging for `Debian` there is also a special mailing list[3]. Just reading along this list is useful.

[3] `https://lists.debian.org/debian-mentors/`

Chapter 8

Checking the sources

Having selected the software to be packaged for Debian, the source code must now be examined in more detail. The goal of this examination is to determine that the selected software can be packaged by the package maintainer for Debian-Main. To do this, it must be verified that **all** parts of the source code conform to the *Debian Free Software Guidelines* (DFSG)[6] and are consistent with the Debian policy[7] .

This requires an intensive examination of **all** source code files.

8.1 License verification

In Debian, only Free Software as defined in the *Debian Free Software Guidelines* [6] may be published in the *main* branch. It should be noted that the *Debian Free Software Guidelines (DFSG)* apply not only to software licenses, but according to clause 1 of version 1.1 of the social contract ("'all components'") also to licenses of images, sounds, texts, etc.

Checking this requirement has a big part in the work of a package maintainer, especially to release a new package in Debian.

The result of this work is then used in the creation and maintenance of the *debian/copyright* file. Thus this file is also one of the most important files of a Debian package. It describes the copyright situation.

This file must accurately describe the copyright and licenses of all files in a source package using specific syntax (DEP-5).[17]

The *debian/copyright* file is checked by the FTP masters[18] when a new package is to be accepted in the Debian project.

To avoid overlooking entries for this purpose, at least a four-eyes-principle (peer-review)[19] applies.

The contents of the copyright file must therefore accurately reflect the licenses of all files. The license is often specified in the comments of a source file.

The entire source code must be tested from this point of view. There are some tools that help the package maintainer to perform these tests. [1]

[1] https://wiki.debian.org/CopyrightReviewTools

8.1.1 *debmake*

The *debmake -cc* command is also used by the build script to create the *debian/copyright* file (Chapter **??**, page **??**).

In practice, it turns out that this is sometimes not sufficient. This is because the information about the licenses and the authors is not stored following a standard. Therefore the program *debmake* does not find all entries concerning the required *copyrights*. So further tests are needed .

8.1.2 *licensecheck*

The *licensecheck* program (from the package of the same name) is able to scan source files and reports the copyright and licenses specified in them. However, it does not summarize this information: A copyright line is generated for each file in a package.

The test was performed with the command line

```
licensecheck --check '.*' --recursive --deb-machine --lines 0 *
```

as detailed in the wiki[2].

Unfortunately, the manual page (man page) of *licensecheck* is rather unproductive. More information is provided by the command:

```
licensecheck --help
```

The output must be evaluated manually, since *licensecheck* also tries to display the copyright for so-called binary files (e.g. images, fonts).

8.1.3 *scan-copyrights*

The *scan-copyrights* program from the *libconfig-model-dpkg-perl* package can update an existing copyright file in *d/copyright* by rescanning the source.

The command line for this is:

```
scan-copyrights
```

The program can also create such a file from scratch. This is done in DEP-5 format. [17]

This program is written in `Perl` and uses *licensecheck*.

8.1.4 *licensing*

The *licensing* command from the *licenseutils* package can scan the source code and return found licenses with the command.

```
licensing detect *
```

It can also add license templates to new code.

[2] `https://wiki.debian.org/CopyrightReviewTools#licensecheck`

8.1.5 *cme*

Another tool is *cme*. *cme* checks and/or edits the data in the configuration files. Among other things, this can be used to check whether the copyright file provided by the original author contains all the necessary information.

The *cme fix dpkg* command checks the *dpkg* files, updates obsolete parameters, and applies any fixes (Chapter **??**, page **??**).

With

```
cme update dpkg-copyright
```

the licenses listed in the headers of the source code files are checked and listed.

With

```
cme check dpkg-copyright -<Path/filename>
```

data can be read from any file.

There is also a graphical user interface for this package with *libcomcnfig-model-tkui-perl*.

8.1.6 Manually

The use of these tools is sometimes not sufficient. It may happen that further authors are named somewhere in the code. This can be searched for with the following commands:

1. grep --recursive --ignore-case “(c)“ .
2. grep --recursive --ignore-case “copyright“ .
3. grep --recursive --ignore-case “author“ .

8.1.7 Stumbling blocks

There are the following stumbling blocks:

1. co-delivered build dependencies
2. microcode
3. co-delivered non-free documents
4. licenses incompletely observed by the upstream author.

Such files must be excluded from the **.orig.tar.xz* to be published, if this is reasonably possible. An appropriate version designation must also be chosen. [20]. In this program script, it is intended to exclude the files using the *debian/copyright* file (chapter **??**, page **??**)

If the source code package contains differently licensed files, the source code package as a whole must be examined in addition to checking the licenses of the individual source code files. It must be checked whether the licenses used are compatible with each other. The *FINOS Open Source License Compliance Handbook*[21] is helpful here.

8.2 Identifying the Programming Language

Software is written in different programming languages. There are also programs written in several programming languages. Depending on the programming language, different compiler and build systems must be used.

There are several build systems for building `Java` packages. This is described in detail in a separate chapter (chapter 11, page 41).

8.3 Checking the Dependencies

In order to release a package in `Debian`-main, **all** dependencies (including build dependencies) must already be available in `Debian`-main[7]. This means that dependencies that are not yet available must be packaged first.

How the dependencies can be determined depends on the programming language. Hints on unfulfilled build dependencies can also be found in the build error messages.

For `Java` programs, there are several places where this information can be found. This is described in the corresponding chapter (chapter 11, page 41).

8.3.1 Identify dependencies with *packages.debian.org*

8.3.2 Search for Dependencies at the Console

With *apt-file find* [3] it can be determined if needed dependencies are already packaged in Debian.

8.4 Modifications of the Source Code

The source code must be carefully checked to see if any changes need to be made to it for the `Debian` package. Such a need can have various causes.

The upstream release may contain parts that do not comply with the `Debian` policy[7] or the `Debian` Free Software Guidelines[6] .

Changes to the source code specifically for the `Debian` package can be made as follows:

- Whole files can be excluded. This is done when building a new version (chapter **??**, page **??**).
- Changes can be made to upstream files by *patching* (chapter 8.4.2, page 30). This is done by the program script when building a new revision (chapter **??**, page **??**).

8.4.1 Exclude complete files

If entire files need to be excluded, a source code package must be created that no longer contains these files.

[3] `https://manpages.debian.org/unstable/apt-file/apt-file.1.en.html`

Before *mk-origtargz* is called, the program script allows individual source code files to be excluded from inclusion in the *orig* archive.

The files that do not comply with the `Debian` policy[7] or the `Debian` Free Software Guidelines (`DFSG`)[6] must be removed.

A new version must always be built, even if no new upstream source code archive is used.

The exclusions should be documented with their rationale in the *debian/README.source* file (chapter **??**, page **??**).

8.4.1.1 Listing of files to be excluded

The files to be excluded can be listed either in a separate file or in the *debian/copyright* file in *DEP-5 format* [17].

Example:

```
Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/ Upstream-Name: <UpstreamName> Source: <URL
```

In the enumeration, the files to be excluded are separated by one character using spaces or line breaks with indentation.Case distinctions

8.4.1.2 Case distinctions

Often it is already known before the package is submitted to the *New Queue* that files are to be excluded. Since there is no *debian/copyright* file at this point, it is a good idea to list the files to be excluded in a separate file.

In response to the corresponding question in the program script, specify that files are to be excluded. Then the program script determines where to store the information to exclude files. This is a separate file before submitting the first package of a new project..

This is done by creating a file *<SourceName>-excluded* in the directory *PrjPath* (chapter **??**, page **??**).

Die Informationen in dieser Datei sind bei der Erstellung der Datei *debian/copyright* dorthin zu übertragen (Kapitel **??**, Seite **??**).The information in this file is to be transferred there when the *debian/copyright* file is created (chapter **??**, page **??**).

After a rejection from the `New Queue`, the package must be re-provisioned for release. Again, files whose license is not `DFSG` compliant may have to be excluded. The exclusion of the files already applies to the *.orig.tar.(gz | bz2 | xz) to be published. Usually, however, no other (newer) upstream version is available at this time. However, the version to be created must be larger than the previous version but smaller than the next expected upstream version..

In the case of removing files that are not DFSG compliant, this is the *+dfsg* attachment that directly follows the upstream version. If this attachment already exists, it is incremented after *+dfsg1* (see chapter 8.4.1.3, page 30). Thus, a new tarball with a larger version label can be created with *mk-origtargz*. (Chapter 9.1, page 33)

The names of the files to be excluded are added to the then already existing file *debian/copyright* in DEP-5 format[17].

After committing these changes in the *debian/* directory, a new version can be built. For this the previous upstream archive is used.

In this way, a new orig tarball is created with *mk-origtargz* without the files to be excluded from the previous *.tar.gz and its contents are inserted into the existing Git repository with *gbp import-orig* (chapter **??**, page **??**).

If a new upstream version requires files to be excluded, or if the files to be excluded change, proceed in the same way as just explained. The copyright file must be changed and a corresponding commit must be executed. Again, when building the new version, specify that files are to be excluded.

8.4.1.3 Naming of packages when excluding files

So that the version designation of this new version is larger than that of the previous version, but smaller than the expected next upstream version, the previous version designation is provided with an appendix. This makes clear at the same time that and why the upstream code was changed.

Common additions to the version string are *+dfsg* and *+ds*[4].

If licenses of individual files of the upstream source code do not satisfy the `Debian Free Software Guildelines (DFSG)` and therefore should not be published as sources by `Debian`, the supplement*+dfsg* is used. If other reasons require the exclusion, one takes *+ds* ("Debian Source"). [20]

This is because it is possible that initially only files that do not comply with the `Debian` policy were excluded.

If it is later determined that the uploaded original archive still contains non-free documents, this then leads to a version designation according to the pattern *<version number>+ds+dfsg.*[5]

In some cases, a plus sign (+) can cause problems especially when building Java packages.

The naming of the `Debian` packages shown also has consequences for the contents of the *debian/watch* file (chapter 9.2 page 34 and chapter **??** page **??**).

8.4.2 Changes in individual source code files (patching)

Changes to source code filesare necessary when adjustments have to be made due to different build environments. Other cases are bug fixes, especially *security patches*.

Die klassische Methode ist die im Folgenden beschriebene mit *quilt*.

Daneben kann dies auch in einem *Patch-Queue*-Zweig mit *gbp pq* erfolgen (Kapitel 8.4.2.2, Seite 31)

[4] `https://wiki.debian.org/DebianMentorsFaq` in section *What does "'dfs"' or "'ds"' in the version string mean*

[5] s. Mentors-FAQ 2.6[20]

8.4.2.1 Patching with *Quilt*

Änderungen an Quellcode-Dateien erfolgen durch sogenannten Patch-Dateien. Diese dokumentieren den ursprünglichen Quellcode und die jeweiligen Änderungen. Eine weitere Datei (*debian/patches/series*) steuert die Reihenfolge der Anwendung der Patches. Herkömmlicherweise geschieht dies mit *quilt*. *Dquilt* ist eine `debian`-spezifische Anpassung für *quilt*.

Hierzu gibt es eine Beschreibung im *Debian-Leitfaden für Neue Paketbetreuer*[6] und im *Debian-Wiki*[22]. Daneben gibt es eine allgemeine Einführung in die Nutzung von Quilt [23].

In Kapitel 16.6 (Seite 68) wird beschrieben, wie die `debian`-spezifische Anpassung erzeugt wird.

8.4.2.2 Patching in a *Patch-Queue*-Branh

Da ein Arbeitsablauf mit `Git` bzw. *git-buildpackage* [3] zum Bauen der Debian-Pakete genutzt wird, bietet es sich an, *gbp pq* statt *quilt*[7] einzusetzen.

Die Änderungen erfolgen dann in einem eigenen `Git`.Zweig

Bei der Nutzung von *gbp pq* erfolgen in diesem Zweig dann *alle* Codeänderungen. Diese Änderungen sollten thematisch und kleingliedrig sein. Dann können die Patches einfacher nach Upstream fließen oder von anderen Distributionen übernommen werden. Dies erleichtert auch die Anpassung an eine neue Upstream-Version.

Die Grundidee ist, dass Patches vom `Debian`-Zweig in den Patch-Queue-Zweig so importiert werden, dass jeweils eine Patchdatei in *debian/patches* jeweils einem Commit auf dem Patch-Queue-Zweig entspricht.

Die Anlage des Patch-Queue-Zweiges erfolgt mit *gbp pq import*. (s. a. Kapitel **??** Seite **??**)

Der erstellte Zweig wird nach dem Zweig benannt, aus dem er importiert wurde. Zur Unterscheidung wird *patch-queue/* vorangestellt. Wird also die `Debian`-Paketierung auf *debian/sid* gemacht und ein *gbp pq import* ausgeführt, wird der neu erstellte Zweig *patch-queue/debian/sid* genannt.

Das Programm-Skript ermöglicht diesen Import vor dem Herunterladen einer neuen Upstream-Version (Kapitel **??**, Seite **??**) und als Auswahlmöglichkeit zum Patchen (Kapitel **??**, Seite **??**).

Im Patch-Queue-Zweig kann mit den bekannten `Git`-Befehlen (*rebase*, *commit* und *–amend*, etc.) an den Commits gearbeitet werden. Wenn dies erledigt ist, wird *gbp pq export* verwendet, um die Commits auf dem *Patch-Queue-Zweig* wieder in Patches in *debian/patches/*-Dateien umzuwandeln

(Cahpter **??**, page **??**).

Dieser beschriebene Workflow erleichtert z. B. das Cherry-Picking von Patches für stabile Releases, die Weiterleitung von Patches an neue Upstream-Versionen durch die Verwendung von *git rebase* auf den Patch-Queue-Zweig (bereits angewandte Patches werden automatisch erkannt) sowie das

[6] `https://www.debian.org/doc/manuals/maint-guide/modify.en.html`

[7] section gbp.patches.html [3]

Neuordnen, Ablegen und Umbenennen von Patches, ohne auf *quilt* zurückgreifen zu müssen. Die erzeugten Patches in *debian/patches/* haben alle notwendigen Informationen, um sie nach Upstream weiterzuleiten, da sie ein Format ähnlich dem *git-format-patch* verwenden.

Der Hauptnachteil dieses Arbeitsablaufes ist der Mangel an Historie im Patch-Queue-Zweig, da er häufig fallen gelassen und neu erstellt wird. Aber es gibt natürlich eine vollständige Historie auf dem `Debian`-Zweig im Verzeichnis *debian/patches/*.

Zu beachten ist, dass *gbp pq* derzeit keine vollständige Unterstützung für DEP3-Header[24] bietet.

Zunächst wird versucht, mit git-mailinfo[25] zu parsen, was nur die Felder *From* und Subject unterstützt. Wenn keines dieser beiden vorhanden ist, wird *gbp pq* versuchen, den Patch vom DEP3-Format in ein git-mailinfo(1)-kompatibles Format zu konvertieren. Dabei wird zuerst aus den Feldern “Autor” und “Betreff” über die erste Zeile des Feldes Beschreibung geladen. Anschließend werden alle zusätzlichen Felder (wie “Origin” und “Forwarded”) und der Rest der Beschreibung (falls vorhanden) an den *mail body* angehängt.

Chapter 9

Versioning of the packages

Sowohl die Software, die für Debian gebaut werden soll, als auch `Debian`-Pakete haben Versionsbezeichnungen. Dabei sollte die Versionsbezeichnung der Upstream-Software in der des Debian-Paketes enthalten sein.

Die Namen aller Debian-Binärpaketdateien sind folgendermaßen aufgebaut: <foo>_<Versionsnummer>-<Debian-Revisionsnummer>_<Debian-Architektur>.deb. [1]

9.1 Versionierungsschema

Die Versionierung der Pakete folgt einem fest definierten Schema. Damit wird sichergestellt, dass alle Werkzeuge, die diese Versionierung verwenden, auf die gleiche Systematik zugreifen können. Dabei muss sichergestellt werden, dass das System die neue Versionsbezeichnung auch als größer als die vorherige Versionsbezeichnung erkennt. (s.a Kapitel **??**, Seite **??**)

Wie muss also die Versionsbezeichnung des zukünftigen Paketes zusammensetzt sein? Nach der Versionsbezeichnung des Upstream-Paketes können debianspezifische Zusätze erfolgen. Diese können anzeigen, ob und warum Teile der Upstream-Sortware entfernt wurden oder ob das Paket auf ein älteres Paket zurückportiert wurde. Außerdem ist in der Versionsbezeichnung des Paketes eine `Debian`-Revisionsnummer enthalten. Diese Revisionsnummer wird insbesondere dann erhöht, wenn das Paket wegen Korrekturen in Dateien im Verzeichnis *debian/* neu gebaut wurde.

Mögliche Varianten können mit dem folgenden Befehl auf ihre Tauglichkeit überprüft werden:

```
dpkg --compare-versions Version1 Operant Version2 &&\echo "OK"
```

As operands these are to be used: lt le eq ne gt
lt kleiner als (<)
le kleiner oder gleich (<=)
eq gleich (=)

[1] DebianFAQ2019, Chapter 7.3[15]

ne ungleich (!=)
ge größer oder gleich (>=)
gt größer als (>)

Die Versionsvergleichsregeln können wie folgt zusammengefasst werden:[2]

- Zeichenketten werden von Anfang bis Ende verglichen.
- Buchstaben sind größer als Ziffern.
- Ziffern werden als Ganzzahlen verglichen.
- Buchstaben werden in ASCII-Sortierreihenfolge verglichen.
- Es gibt besondere Regeln für Punkt (.), Plus- (+) und Tilde- (~) Zeichen, wie folgt: *Beispiel:* 0.0 < 0.5 < 0.10 < 0.99 < 1 < 1.0~rc1 < 1.0 < 1.0+b1 < 1.0+nmu1 < 1.1 < 2.0

In some cases, a plus sign (+) can cause problems especially when building Java packages.

Die "richtige" Versionierung spielt eine Rolle, wenn diese von Programmen gelesen und genutzt werden soll. Dies sind vor allem die Programme *dpkg*, *apt* und *uscan*.

9.2 *uscan*and the file *debian/watch*

uscan ist ein nützliches Werkzeug im `Debian`-Projekt. Mittels *uscan* kann geprüft werden, ob es eine Upstream-Version gibt, die aktueller (also höher versioniert) ist, als das aktuelle `Debian`-Paket.

Das Programm *uscan* wird mit dem Paket *devscripts* zur Verfügung gestellt.

Anhand der in dieser Datei enthaltenen Daten prüft *uscan*, ob neuere Versionen in der Originalquelle vorhanden sind (Kapitel **??**, Seite **??**), und kann sie gegebenenfalls herunterladen (Kapitel **??**, Seite **??**).

Die durch *uscan* gewonnene Informationen werden auch auf der Seite des jeweiligen Maintainers
`https://qa.debian.org/developer.php?<Maintainers>@debian.org`
dargestellt.

Dafür benötigt *uscan* vor allem eine taugliche Datei *debian/watch*. Die Erstellung dieser Datei, die auch beim Herunterladen des Quellcodes mit *uscan* (Kapitel **??**, Seite **??**) benptigt wird, wird vom Programm-Skript unterstützt (Kapitel **??**, Seite **??**).

Wenn Dateien ausgeschlossen werden und die **.orig.tar.xz*-Datei entsprechend benamt wurde (Kapitel 8.4.1.3, Seite 30), sind entsprechende Optionen in die Datei *debian/watch* einzufügen. Diese Optionen sind *repack*, *compression*, *repacksuffix* und *dversionmangle*. Dann kann bei der nächsten neuen Upstream-Version das Herunterladen mit *uscan* durchgeführt werden. Dies gilt auch für die Prüfung mittels *uscan*, ob es eine neue Upstream-Version gibt.

Beispiel:

[2] New Maintainer Guide, Kapitel 2.6 [11, Kapitel 2.6] s.a. Debmake-doc, Kapitel 5.2

```
opts=repack,\ compression=xz,\ repacksuffix=+dfsg,\ dversionmangle=s/\+dfsg// \
```

9.3 *apt*

Die korrekte Versionierung des `Debian`-Paketes ist dann nicht ganz einfach, wenn nicht für `Debian`-Sid gebaut wird (Kapitel 18.7, Seite 81).

dpkg bzw. *apt* müssen zutreffend ermitteln können, ob ein neueres Paket im jeweiligen Release-Zweig zur Verfügung steht.

Chapter 10

dh_make

Es gibt erfreulicherweise viele nützliche Programme (Hilfsprogramme), die das Bauen von `Debian`-Paketen erleichtern und vereinheitlichen. Wer sich mit dem Bauen von `Debian`-Paketen befasst, wird immer wieder auf das Werkzeug *dh_make* stoßen. Das Paket heißt *dh-make* (mit Bindestrich).

dh_make zeigt auf, wie ein `Debian`-Paket potenziell aussehen kann. Dies ermöglicht das Erlernen des Paketierens an konkreten Beispielen. Lehrreich sind vor allem die im Programmpaket enthaltenen Vorlagen (Templates).

dh_make erstellt viele Dateien in dem Verzeichnis *debian/*. Für die meisten Pakete wird nur eine Teilmenge davon benötigt.

Die Praxis zeigt, dass jedes Mal geprüft werden muss, welche Dateien für das konkrete Paket nicht benötigt werden. Auch sind die erzeugten Dateien unvollständig und erfordern einiges an Handarbeit.

Um *dh_make* nutzen zu können, sind einige Vorbereitungen zu treffen. Dazu wird ein entsprechendes "‘Arbeitsverzeichnis"’ angelegt. Darin wird das heruntergeladene Quellcode-Archiv abgelegt. Bevorzugt wird ein hierfür vorgesehenes Tar-Archiv.

Dieses Tar-Archiv wird dort entpackt. Dies kann auf der Konsole mit

```
tar --extract --auto-compress --file=<Tar-archiv>.tar.gz
```

durchgeführt werden.

dh_make muss in dem Verzeichnis aufgerufen werden, das den Quellcode enthält. Dabei kann es vorkommen, dass der Name des Verzeichnisses mit dem Quellcode außer Kleinbuchstaben noch weitere Zeichen enthält. *dh_make* benötigt jedoch einen Namen, der Einschränkungen unterliegt und dem Schema <paketname>-<version> entspricht.

Damit *dh_make* die Vorlagen korrekt füllen kann, wird in solchen Fällen an *dh_make* die zusätzliche Option *–package-name* hinzugefügt.Hiermit kann die Verwendung des nachfolgenden Namens als Name des Quellcodepaketes erzwungen werden. Dieser Verzeichnisname muss die Versionsbezeichnung enthalten. Zwischen Paketnamen und Versionsbezeichnung muss ein Bindestrich stehen.

So kann nach dem Wechsel in dieses Verzeichnis dort mit

```
dh_make --createorig --packagename <SourceName>
```

der Source-Tar-Ball in der von `Debian` gewünschten Form *<SourceName>.orig.tar.gz/xz* erzeugt werden. Daneben wird ein Entwurf der Dateien im Verzeichnis *debian/* erstellt.

Dabei werden folgende Informationen abgefragt.

Type of package = Paketklassen single, indep, library, python mit folgender Bedeutung

single Es wird ein einzelnes Binary-Paket (*.deb) erstellt. Dies ist der Normalfall

indep Es wird ein Binary-Paket erstellt, das von der Architektur unabhängig ist.

library Es werden mindestens zwei Binärdateien erstellt. Ein Bibliothekspaket, das nur die lib in /usr/lib enthält, und ein weiteres *-dev_*.deb-Paket, das die Dokumentation und C-Header enthält.

python

Einzelheiten zun Paket Es werden die ermittelten Werte für *Maintainer Name*, *Email-Address*, *Date*, *Package Name*, *Version*, *License* und *Package Type* zur Bestätigung angezeigt.

Es werden folgende Dateien erzeugt. Dabei werden zunächst die Dateien gelistet, die auf jeden Fall benötigt werden, jeweils in alphabetischer Reihenfolge. Die Dateien *README.Debian* und *README.source* können zu Informationszwecke genutzt werden. Zum Schluss folgen solche Vorlagen, die nur in sehr wenigen Paketen genutzt werden.

- changelog
- control
- copyright
- rules
- salsa-ci.yml.ex
- source/format
- watch.ex
- README.Debian
- README.source
- manpage.1.ex
- manpage.sgml.ex
- manpage.xml.ex
- postinst.ex
- postrm.ex
- preinst.ex
- prerm.ex
- <packagename>.cron.d.ex
- <packagename>.doc-base.EX
- <packagename>-docs.docs

Die nicht benötigten Dateien werden aus dem Verzeichnis *debian/* entfernt.

Dass der Verzeichnisname die jeweilige Versionsbezeichnung enthalten muss, ist bei der Verwendung von *git* ungünstig. Daher wird bisher davon abgesehen, dieses Hilfsprogramm einzusetzen.

Chapter 11

Java-Pakete bauen

Das Bauen von Java-Paketen weist besondere Herausforderungen auf. Diese Besonderheiten werden im Folgenden ausführlich beschrieben.

Für das Packen von Java-Paketen gibt es eine speziellen Richtlinie, die Debian-Java-Policy[26].

Weitere Informationen erhält man auch aus der Java-FAQ[27], vorallem im Abschnitt 2.4[1]

11.1 Herausforderungen

In einem Blog-Artikel[28] beschreibt Hans-Christoph Steiner zutreffend die Herausforderungen für ein Debian-Paket in Bezug auf Java-Software.

Das Debian-*Java*-Team muss demnach konsequent gegen die *Java*-Standardpraxis ankämpfen, alle Abhängigkeiten in einer einzigen **.jar*-Datei zu bündeln. Dies bedeutet, dass es keine gemeinsamen Sicherheitsaktualisierungen gibt. Jeder Entwickler muss jede Abhängigkeit in diesem Modell selbst aktualisieren. Das funktioniert hervorragend für große Unternehmen mit Mitarbeitern, die sich dieser Aufgabe widmen.

Für die Mehrheit der Debian-Anwendungsfälle funktioniert das schlecht. Debian hält das Versprechen ein, dass die Leute *foo* einfach passend installieren können, damit es funktioniert, und Sicherheitsaktualisierungen erhalten. Der Benutzer muss nicht einmal wissen, in welcher Sprache das Programm geschrieben ist, es funktioniert einfach.

Die Hoffnung, dass die *Java*-Entwickler-Gemeinschaft sich den Wert dieser Anwendungsfälle zu eigen macht und Debian hilft, indem sie es einfacher macht, *Java*-Projekte in der Standard-Distributionsmethode zu paketieren, mit gemeinsamen Abhängigkeiten, die unabhängig aktualisiert werden, hat sich bislang nur rudimentär erfüllt.

Für das Packen der Java-Pakete gibt es weitere Dokumente, die Hilfestellungen dazu geben. Dies ist speziell die Debian-Java-Policy[26]. Ferner gibt es noch eine Beschreibung über das Paketieren mit den Javatools[29]. Darin

[1] https://www.debian.org/doc/manuals/debian-java-faq/ch2.en.html#s2.4

werden die Java-Helper beschrieben. Dieses Tutorial ist auch Bestandteil des Debian-Paketes *javahelper*. Zusätzlich gibt es noch die Java-FAQ[27]

11.2 Anwendungen und Bibliotheken

Es gibt zwei Kategorien von Java-Paketen: Anwendungen und Bibliotheken. Sowohl Java-Anwendungen als auch Java-Bibliotheken können im Quellcode (weitere) Bibliotheken enthalten, die vorkompiliert sind.

Sollen in Java geschriebene Anwendungen paketiert werden, müssen in der Regel auch immer Java-Bibliotheken als Debian-Pakete bereitgestellt werden.

Sowohl Anwendungen als auch Bibliotheken werden in **.jar*-Dateien als *Zip*-Archiv bereitgestellt.

Beide Arten müssen grundsätzlich als Java-Bytecode (*.class-Dateien, verpackt in einem *.jar-Archiv) und mit einer "'*Architecture: all*"' (Kapitel **??**, Seite **??**) ausgeliefert werden.

Hat man von einer zu paketierenden Anwendung oder Bibliothek zunächst nur ein (kompiliertes) **.jar*-Archiv, kann ein Blick in die Datei *MANIFEST.MF* des jeweiligen *.jar*-Archivs helfen, diejenige *URL* zu ermitteln, unter der der Quellcode veröffentlicht worden ist.

11.2.1 Java-Programme paketieren

Java-Programme sind dazu bestimmt, von Endbenutzern ausgeführt zu werden. Diese benötigen auch eine Manpage, soweit es sich um selbstständig ausführbare Programme handelt. [2]

Das Paket muss auch sicherstellen, dass die richtige Klasse als Hauptklasse verwendet wird.

Zusätzliche Klassen im Paket müssen in ein oder mehrere Java-Archiv(e) (**.jar*-Datei(en) gepackt werden, die in */usr/share/java* (wenn sie für die Verwendung durch andere Programme vorgesehen sind) oder in ein "'privates"' Verzeichnis in */usr/share/<package>* abgelegt werden können.

11.2.2 Java-Bibliotheken paketieren

Java-Bibliotheken dienen der Erfüllung von Abhängigkeiten von Programmen. Dies können Build- und/oder Laufzeitabhängigkeiten sein.

11.3 Abhängigkeiten bei Java-Paketen

Java-Anwendungen hängen immer auch von Java-Bibliotheken ab. Außerdem wird für die Java-Pakete immer *default-jdk* als Abhängigkeit benötigt. Dahinter verbirgt sich in der Regel die aktuell verfügbare OpenJDK-Version,

[2] Kapitel 2.3 der Java-Policy[26]

eine freie Java Implementierung mit Langzeitunterstützung (JDK = Java Development Kit).

Daneben gibt es spezielle Abhängigkeiten, die sich aus dem Buildsystem ergeben. Diese werden bei den einzelnen Buildsystemen im Folgenden besprochen.

11.3.1 Weitere Abhängigkeiten feststellen

Java-Entwickler fügen ihren Quellcode-Paketen die benötigten Build-Abhängigkeiten - auch Bibliotheken von Drittanbietern - in kompilierter Form als *.jar-Archive hinzu.

Gerade unter Java-Entwicklern ist es leider sehr verbreitet, auf irgendwo im World Wide Web veröffentlichte Bibliotheken zurückzugreifen und diese dann vorkompiliert mit auszuliefern. Diese Bibliotheken müssen durch in Debian veröffentlichte Pakete ersetzt werden.

Zur Feststellung von Abhängigkeiten ist also im Quellcode nach **.jar*-Archiven zu suchen.

In diesem Falle ist für den Debian-Maintainer ein Repacking zum Entfernen dieser Dateien aus dem Upstream-Tarball notwendig. (Kapitel **??**, Seite **??**)

Die ausgeschlossenen Bibliotheken müssen durch eigenständige Pakete verfügbar gemacht werden.

11.3.2 Abhängigkeiten ermitteln

Hat man Abhängigkeiten festgestellt, so ist zu ermitteln, wie diese Abhängigkeiten erfüllt werden können. Hierzu ist als erstes zu prüfen, ob bereits entsprechende Debian-Pakete existieren. Um zu ermitteln, ob eine Abhängigkeit bereits in Debian paketiert wurde, gibt es mehrere Wege.

Debian-Wiki (für Maven-Pakete):
https://wiki.debian.org/Java/MavenPkgs

11.4 Buildsysteme für Java-Pakete

Zum Bauen von Java-Paketen werden verschiedene Buildsysteme genutzt. Hierbei handelt es sich um *maven*, *ant* und *gradle*. Daraus ergeben sich Eigenheiten, die beim Paketieren für Debian beachtet werden müssen. Es gibt auch Java-Pakete, die ohne eines dieser Buildsysteme gebaut werden. Die folgenden Beschreibungen orientieren sich an den gemachten Erfahrungen der Autoren.

11.4.1 Das Buildsystem *maven*

Apache Maven ist ein Werkzeug zur Verwaltung und zum Verstehen von Softwareprojekten.

Ein wesentliches Erkennungsmerkmal für das `Maven`-Buildsystem ist die Datei *pom.xml*. Diese enthält alle Informationen zum jeweiligen Softwareprojekt und folgt einem standardisierten *XML*-Format.

Wesentlich ist auch die Standard-Verzeichnisstruktur eines `Maven`-Projektes, die im Folgenden dargestellt wird.[30]

src/main/java Anwendungs- / Bibliotheksquellen
src/main/resources Anwendungs- / Bibliotheksressourcen
src/main/filters Ressourcenfilterdateien
src/main/webapp Webanwendungsquellen
src/test/java Testquellen
src/test/resources Ressourcen testen
src/test/filters Testen Sie Ressourcenfilterdateien
src/it Integrationstests (hauptsächlich für Plugins)
src/assembly Assembly-Deskriptoren
src/site Website
LICENSE.txt Lizenz des Projekts
NOTICE.txt Hinweise und Zuordnungen für Bibliotheken, von denen das Projekt abhängt
README.txt Readme des Projekts
pom.xml Beschreibung des Projektes und der Konfiguration

In den meisten Fällen reicht es aus, den Zweig *src/main/* als `Debian`-Paket zu bauen. Dies gilt besonders dann, wenn Bibliotheken gebaut werden, die als Abhängigkeiten für Anwendungen benötigt werden.

Das Upstream-Paket muss bestimmte Voraussetzungen erfüllen, um mit dem Buildsystem `Maven` gebaut werden zu können.

Dazu gehört auf jeden Fall mindestens eine *pom.xml*-Datei. Diese Datei kann an verschiedenen Stellen innerhalb des Source-Codes liegen. Manchmal wird sie - besonders im `Maven`-Repositorium (`https://mvnrepository.com/`.) - nur zusätzlich bereitgestellt.

Wenn vorhanden, ist es jedoch oft besser, `Github` oder andere Git-Repositorien als Quelle zu wählen. Dabei ist auf die Version zu achten! Für manche Pakete wird der Quellcode auch unter der Domain `https://gitbox.apache.org/repos/asf` veröffentlicht.

11.4.2 Paketieren mit *maven*

Für *maven*-basierte Pakete wird die Verwendung von *maven-debian-helper*[3] empfohlen.

In der Datei *debian/rules* (Kapitel **??**, Seite **??** wird

Zur Unterstützung des Bauens mit *maven* enthält es folgende Befehle:

mh_genrules(1) generiert, zumindest teilweise, die Datei *debian/rules*
mh_lspoms(1) sucht nach allen POM-Dateien, die im Quellcode des Projektes angegeben sind.
mh_make(1) generiert das `Debian`-Paket durch Lesen der Informationen aus der Maven-POM

[3] `https://manpages.debian.org/unstable/maven-debian-helper/index.html`

mh_resolve_dependencies(1) löst die Abhängigkeiten auf und erzeugt die Datei *<Paketname>.substvars*, die die Liste der abhängigen Pakete enthält, zur Nutzung von *debian/control*.

Zu diesen Befehlen gibt es auch entsprechende *Man-Pages*.

Hiervon wird zunächst nur der Befehl *mh_make*[4] im Plugin *build-gbp-maven-plugin.sh* genutzt. (Kapitel **??**, Seite **??**)

Bei der Nutzung von *mh_make* ist zu beachten, dass in der Umgebung, in der *mh_make* gestartet wird, alle Abhängigkeit des zu bauenden Paketes vorhanden sein sollten. Andernfalls sind etwaige Einträge in den einschlägigen Dateien (z.B. *debian/control*) manuell nachzuholen. (Kapitel **??**, Seite **??**)

Dazu empfiehlt es sich, diese Operationen in einer eigens dafür eingerichteten *chroot* durchzuführen, in der alle Abhängigkeiten installiert werden können, ohne das eigentliche Host-System zu belasten. Die Einrichtung derselben ist in Kapitel 16.5 (Seite 67) beschrieben.

Von *mh_make* werden folgende Dateien im Verzeichnis *debian/*angelegt:

- maven.cleanIgnoreRules
- maven.rules
- maven.ignoreRules
- maven.properties
- <Paketname>.poms
- maven.publishedRules

Zusätzlich werden von *mh_make* auch die (Standard-)Dateien

- changelog
- control
- copyright
- rules
- README.source

erstellt. Dies ist bei den weiteren Arbeiten an diesen Dateien zu berücksichtigen (Kapitel **??**, Seite **??**).

Mit *apt-file find*[5] kann festgestellt werden, ob es zu einer *artifactID* ein Debian-Paket gibt. Diese Funktion wird auch von *mh_make* genutzt. Die *artifactID* befindet sich in der *pom.xml*-Datei aus dem Upstream-Code.

Unter `wiki.debian.org/Java/MavenPkgs/Unstable` befindet sich eine Liste aller in `Debian` bereits paketierten *Maven*-Pakete. [6]

mh_make erstellt aus der *pom.xml* auch die Datei *debian/<Paketname>.poms*.

In dieser Datei werden alle im Paket vorkommende *pom.xml*-Dateien mit ihrem relativen Pfad aufgelistet. Diese können dann dort mit den aufgelisteten Optionen versehen werden.

---

[4] urlhttps://manpages.debian.org/unstable/maven-debian-helper/mh_make.1.en.html

[5] `https://manpages.debian.org/unstable/apt-file/apt-file.1.en.html`

[6] Dank an Thorsten Glaser und der Firma Tarent `https://www.tarent.de/` für diese Unterstützung

46a ⟨*debian/JavaPackage.poms* 46a⟩≡

```
# List of POM files for the package # Format of this file is: # <p
to pom file>[option]* # where option can be: # --ignore: ignore this POM and its artifact if any
don't install the POM. To use on POM files that are created # temporarily for certain artifacts
as Javadoc jars. [mh_install, mh_installpoms] # --no-parent: remove the <parent>tag from the POM
an alternative package to use when installing this POM # and its artifact # --has-package-versio
indicate that the original version of the POM is the same as the upstream part # of the version
package. # --keep-elements=<elem1,elem2>: a list of XML elements to keep in the POM # during a c
operation with mh_cleanpom or mh_installpom # --artifact=<path>: path to the build artifact asso
with this POM, # it will be installed when using the command mh_install. [mh_install] # --java-l
the jar into /usr/share/java to comply with Debian # packaging guidelines # --usj-name=<name>: n
use when installing the library in /usr/share/java # --usj-version=<version>: version to use whe
the library in /usr/share/java # --no-usj-versionless: don't install the versionless link in /us
# --dest-jar=<path>: the destination for the real jar. # It will be installed with mh_install. [
# --classifier=<classifier>: Optional, the classifier for the jar. Empty by default. # --site-xm
Optional, the location for site.xml if it needs to be installed. # Empty by default. [mh_install
```

Eine häufig verwendete Zeile ist *pom.xml –no-parent –has-package-version*. Dies bedeutet, dass das <parent> Tag beim Bauen aus der POM entfernt wird. Die Option *–has-package-version* gibt an, dass die Originalversion der POM dieselbe sei, wie der Upstream-Teil der Paketversion.

Diese Datei kann innerhalb des Bauprozesses einer neuen Debian-Revision angepasst werden. (Kapitel **??**, Seite **??**).

Liegt die *pom.xml* im Wurzelverzeichnis des Quellcodes, werden durch *mh_make* die weiteren Dateien für das Verzeichnis *debian/* erstellt.

Im Maven-Repositorium wird die Datei *pom.xml* oft nur separat bereitgestellt. Diese Datei wird dann im *debian/* Verzeichnis abgelegt.

Daher sind hier einige Einträge in *debian/rules* (Kapitel **??**, Seite **??** erforderlich:

Nach dem Aufruf aller *debhelper* (*dh $@*) wird in einem *override* zunächst das *dh_auto_build* außer Kraft gesetzt, um noch einige vorbereitenden Konfigurationen durchzuführen.

Dazu gehört auch das Kopieren von Dateien an den Ort, wo das Buildsystem diese Dateien benötigt, um das gewünschte Programm zu bauen.

46b ⟨*maven-build* 46b⟩≡

```
override_dh_auto_build: cp debian/pom.xml . dh_auto_b
```

Weitere Literatur

1. https://wiki.debian.org/Java/MavenBuilder
2. https://wiki.debian.org/Java/MavenRepoSpec
3. https://wiki.ubuntu.com/JavaTeam/Specs/MavenSupportSpec

Mit *apt-file find*[7] kann festgestellt werden, ob es zu einer *artifactID* ein Debian-Paket gibt. Diese Funktion wird auch von *mh_make* genutzt. Die *artifactID* befindet sich in der *pom.xml*-Datei aus dem Upstream-Code.

Unter wiki.debian.org/Java/MavenPkgs/Unstable befindet sich eine Liste aller in Debian bereits paketierten *Maven*-Pakete. [8]

[7] https://manpages.debian.org/unstable/apt-file/apt-file.1.en.html

[8] Dank an Thorsten Glaser und der Firma Tarent https://www.tarent.de/ für diese Unterstützung

11.4.3 Paketieren mit *ant*

Das Buildsystem *ant* erkennt man daran, dass es eine Datei *build.xml* gibt. [9]

11.4.4 Paketieren mit *gradle*

11.5 Java-Pakete bauen ohne Buildsystem

Es ist auch möglich , einfach eine Java-Bibliothek aus einem Verzeichnis mit *.java-Dateien zu bauen. Dies ist dann sinnvoll, wenn aus einem größeren Programmpaket nur eine oder wenige einzelne Java-Klasse benötigt wird, um Buildabhängigkeiten anderer Pakete zu erfüllen.

Dabei gibt es für ein solches Java-Verzeichnis kein Buildsystem wie die noch zu beschreibenden *ant* (Kapitel 11.4.3, Seite 47 bzw. *maven* (Kapitel 11.4.1, Seite 43.

47 ⟨*java-builds* 47⟩≡ export DH_VERBOSE=1 export DH_OPTI

```
Class-Path: src/net/numericalchameleon/util/phoneticalphabets.jar export CLASSPATH=/usr/share/

%:
dh $@ --with javahelper --sourcedirectory=src/net/numericalchameleon/util/

override_jh_build:

javac -encoding UTF-8 src/net/numericalchameleon/util/phoneticalphabets/* jh_build jar cvf src
src/net/numericalchameleon/util/phoneticalphabets/*.class

javac -encoding UTF-8 src/net/numericalchameleon/util/spokennumbers/* jh_build
```

Das *javahelper*-Paket bietet Hilfestellung beim Bauen mit *dh*. Diese wird für die Java-Paketierung dringend empfohlen.

Ferner wird die Datei *debian/javabuild* benötigt. Diese enthält zwei Spalten.

Anzugeben ist in der ersten Spalte die zu bauende **.jar*-Datei und in der zweiten Spalte das Verzeichnis des Quellcodes, in dem die **.java*-Dateien liegen.

Beispiel:

```
#NameOfJarFile SourceDirToPackage
```

Das Skript ermöglicht das Erstellen dieser Datei (Kapitel **??**, Seite **??**.

[9] https://wiki.debian.org/Java/Packaging/Ant

Chapter 12

Mozilla-Erweiterungen bauen

Das Bauen von Mozilla-Erweiterungen weist Besonderheiten auf. Diese Besonderheiten ergeben sich aus den verschiedenen Strategien der Installation und Aktualisierung solcher Erweiterungen. Debian-Pakete werden zentral für alle Systemnutzer installiert. Damit können auch zentral die Aktualisierungen durchgeführt werden. Dies hat den weiteren Vorteil, dass alle Nutzer mit den gleichen Versionsständen arbeiten.

12.1 Quellen der Erweiterungen

Unter addons.mozilla.org findet man Erweiterungen für den Firefox veröffentlicht von Mozilla.

Für den Thunderbird werden unter addons.thunderbird.net die Erweiterungen veröffentlicht.

Diese liegen dort als *.xpi*-Archive vor. Sofern dies Quellcode-Pakete sind, gibt es nicht zwingend auch separaten Quellcode-Archive. Zum Bauen des *.orig.tar.xz*-Paketes aus Quellcode-Archiven vom Dateityp *.xpi* wird das Paket *mozilla-devscripts* benötigt (Kapitel 16.1.3, Seite 58).

Manchmal findet man auch Quellcode-Veröffentlichungen von Personen und Projekten beispielsweise auf Github (github.com) oder auf selbstgehosteten Seiten. Diese können auch als *.tar.gz*- oder *.zip*-Archive vorliegen.

12.2 Integration ins Dateisystem

Um alle gewünschten Mozilla-Erweiterungen einheitlich zu bauen, werden die benötigten Teile des Skriptes als *Webext-Plugin* realisiert (Kapitel **??**, Seite **??**). Damit wird auch eine einheitliche Integration in das System gewährleistet.

Ab der Version (>=78) des Thunderbirds können ausschließlich *Mail"=Extensions* genutzt werden. Dies ist ein Einschnitt in der Bereitstellung der jeweiligen

Erweiterungen. Die alten (für TB <= 68.x) Erweiterungen funktionieren nicht mehr.

Die neuen Erweiterungen funktionieren nun ab `Thunderbird` in der Version >= 78.2.

Die Erweiterung muss nun im Verzeichnis */usr/share/webext* mit der Bezeichnung (id) aus der Datei *manifest.json* ergänzt und mit der Endung (.xpi) als Zip-Archiv abgelegt werden (Kapitel **??**, Seite **??**).

Dazu wird die Datei *debian/rules* entsprechend erweitert (Kapitel **??**, Seite **??**). Zusätzlich sind immer die Dateien *<PaketName>.install* (Kapitel **??**, Seite **??**) und *<Paketname>.links* (Kapitel **??**, Seite **??**) so zu erstellen, dass `Thunderbird` die Erweiterung auch findet.

Chapter 13

Python-Pakete bauen

Chapter 14

Metapakete bauen

Metapakete sind solche `Debian`-Pakete, die von einer Gruppe von Paketen abhängen. Damit werden Gruppen von Paketen zusammengefasst, bei denen es sinnvoll ist, sie gemeinsam zu installieren.

Beispiele sind Erweiterungen für eine bestimmre Software. Metapakete weisen einige Besonderheiten auf.

14.1 Kein Upstream-Quellcode

Da es keinen Upstream-Quellcode gibt, benötigt ein Metapaket auch keine Datei *debian/watch*

14.2 Natives `Debian`-Paket

14.2.1 *debian/source/format*

14.2.2 *debian/control*

14.2.3 *debian/rules*

14.2.4 *debian/changelog*

Chapter 15

Konfiguration zur Installation

15.1 *debconf*

Debconf ist ein Konfigurationsmanagementsystem für Debian. Dies dient dazu, Informationen für die Installation vom Nutzer abzufragen und diese dann bei der Installation zu berücksichtigen. [1]

Mit *sudo dpkg-reconfigure debconf* kann *debconf* selbst konfiguriert werden. Dies beinhaltet auch die grafische Oberfläche.

`first step: https://wiki.debian.org/debconf https://wiki.debian.org/debconf`

15.2 *dbconfig-common*

Dbconfig-common stellt eine einfache,zuverlässige und konsistente Methode zur Verwaltung von Datenbanken bereit, die von Debian-Paketen verwendet werden.

[1] `https://wiki.debian.org/debconf`

Chapter 16

System einrichten

Es wird davon ausgegangen, dass bereits ein installiertes und lauffähiges Debian-System existiert. Dabei ist es unerheblich, welcher Zweig (unstable, testing oder stable) darauf läuft.

Zum Signieren der Produkte des Programm-Skripts muss ein *GPG*-Schlüssel des Nutzers auf dem System zur Verfügung stehen. Dieser sollte sinnvollerweise der für das `Debian`-Repositorium vorgesehene Schlüssel sein.

16.1 Abhängigkeiten für das Programm-Skript

Die Pakete, die zum Einsatz des Programm-Skriptes erforderlich sind, sind in den Kopfzeilen desselben aufgeführt (Kapitel **??**, Seite **??**).

Einige dieser Programme werden im folgenden beschrieben

16.1.1 Generelle Abhängigkeiten

Die nachstehend aufgeführten, hilfreichen Programme werden regelmäßig vom Skript zum Bauen eines `Debian`-Paketes eingesetzt:

mk-origtargz *mk-origtargz* benennt den Tarball der Originalautoren um, ändert wahlweise die Komprimierung und entfernt unerwünschte Dateien. Dazu stehen unterschiedliche Optionen zur Verfügung. [31]

gbp import

dh_make *dh_make* kann die benötigten `Debian`-Dateien erstellen. Die Praxis zeigt, dass dies sehr unvollständig ist und einiges an Handarbeit erfordert. Daher wird bisher davon abgesehen, dieses Hilfsprogramm einzusetzen.

debmake -cc

gbp dch Im Programm-Skript wird *dch* über *gbp dch* verwandt. Die Optionen für *dch* können an *gbp dch* durch *–dch-opt= <dch-Optionen>* übergeben werden.

gbp buildpackage

lintian

uscan

debsign

dput
mh_make (Plugin)

16.1.2 Abhängigkeiten für das Bauen von Java-Paketen

Die folgenden Abhängigkeit werden für das Bauen mit Buildsystemen für Java-Pakete benötigt.

58a ⟨*Dependencies1* 58a⟩≡ `# gradle-debian-helper, maven-debian-hel`
`libmaven-bundle-plugin-java,` ⟨*Dependencies5* 58b⟩

16.1.3 Abhängigkeiten für die Mozilla-Erweiterungen

Wird der Quellcode der Mozilla-Erweiterungen nur als **.xpi*-Datei bereitgestellt, muss das folgende Paket auf der Maschine für die Nutzung durch *mk-origtargz* zur Verfügung stehen.
zip Beschreibung wie zip für die Mozilla-Extension.

58b ⟨*Dependencies5* 58b⟩≡ (58a) `# mozilla-devscripts,`

58c ⟨*SBuild* 58c⟩≡ `sudo apt install sbuild schroot debootstrap apt-cache`
`devscripts`

Quelle: `https://wiki.debian.org/sbuild`

16.2 Verzeichnisstruktur

16.2.1 Pfad zu den Projekten

Projekte/Git/01_Salsa

16.2.2 Konfigurationsdateien

16.2.2.1 Für jedes Projekt

Für jedes Projekt wird eine eigene Konfigurationsdatei erstellt. Diese wird im Homeverzeichnis des Nutzers im Verzeichnis *.debian_project/* als Datei *<Projektname>* abgelegt. Darin werden projektspezifische Informationen hinterlegt.

Die Konfigurationsdatei ist technisch ein Shell-Skript, welches von dem Programm-Skript erstellt und geladen wird. Dieses Shell-Skript enthält Variablen, denen dort Werte zugewiesen werden. Die in der Konfigurationsdatei mit Werten versehenen Variablen können durch das Laden vom Programm-Skript verwendet werden.

Ferner enthält die Datei Kommentare. Diese können vom Nutzer beliebig erstellt werden. Ein Teil der Kommentare wird jedoch vom Programm-Skript erstellt. Diese enthalten ein Eigenschafts-Werte-Paar, welches wie die Zuweisung eines Wertes an eine Variable aufgebaut ist. Technischer Hintergrund ist, dass im Eigenschaftsnamen sinnvollerweise Zeichen vorkommen, die im Bezeichner einer Variablen nicht verwendet werden dürfen.

Existiert die Konfigurationsdatei, so wird sie zunächst geladen. (Kapitel **??**, Seite **??**. Man kann sie dann, wenn nötig, editieren. Andernfalls wird diese Datei durch das Skript erstellt. (Kapitel **??**, Seite **??**)

Folgende Informationen werden dort hinterlegt:

SourceName
PackName
ProjectPath = ~/Projekte/Git/01_Salsa
SalsaName
Java-Package
- **SalsaName** = java-team/<SourceName>.git
- **JavaFlag**
- **MavenPluginFlag**
- **MavenPluginPath**
- **Maintainer** =Maintainer=Debian_Java_Maintainers _@lt@pkg-java-maintainers@lists.alioth.debian.org@gt@
- **Uploader** =Mechtilde_Stehmann_@lt@mechtilde@debian.org@gt@

web-Extension-Packages
- **SalsaName** = webext-team/
- **WebextFlag**
- **Maintainer**
- **Uploader**

Python-Packages **SalsaName** = python-team/packages/
WebextFlag
Maintainer
Uploader
DefaultBranch
RecentBranch = debian/sid
RecentUpstreamSuffix = .tar.gz
RecentRepackSuffix = +dfsg | +ds
master_Dist
DownloadUrl
DownloadZip
Maintainer Mechtilde Stehmann <mechtilde@debian.org>
ExcludeFile

16.2.2.2 Für viele Projekte

In der Datei *~/.debian_project/DefaultValues* werden Variablen Werte zugewiesen, die für viele Projekte gelten.

60 ⟨*DefaultValues* 60⟩≡ #!/bin/

```
DefaultProjectPath=/home/<user>/Projekte/Git/01_Salsa HOME=/home/<user>/
```

16.2.2.3 Fingerprint des Maintainer-Schlüssels

Im Verzeichnis *~/.debian_project/* befindet sich auch eine Datei *fingerprint*, die den Fingerprint des Maintainer-Schlüssel enthält. Mit diesem Schlüssel werden die Pakete signiert . (Kapitel **??**, Seite **??**)

16.3 Paketname

Der Paketname muss gänzlich in Kleinbuchstaben geschrieben werden. Die Versionsbezeichnung kann auch Ziffern enthalten. Zusätzlich können +, -, ~enthalten sein.

Die Versionsbezeichnung muss mit einer Ziffer beginnen.

16.4 PBuilder einrichten

Gbp buildpackage verwendet *cowbuilder* . Der Befehl *cowbuilder* ist ein Wrapper für *pbuilder*, welcher die Nutzung eines *pbuilder*-ähnlichen Interfaces in einer *cowdancer*-Umgebung ermöglicht.

16.4.1 *chroot*

Das Paket *pbuilder* enthält Programme, um eine *chroot*-Umgebung aufzubauen und zu betreuen.

Chroot bedeutet *Change root*

Es entspricht guter Praxis, Debian-Pakete in einem *chroot* zu bauen. Damit soll gewährleistet werden, dass das Paket mit den Ressourcen der jeweilige Distribution gebaut werden kann.

Beim Bauen eines Debian-Paketes in dieser *chroot*-Umgebung wird nämlich geprüft, ob die Build-Abhängigkeiten erfüllbar sind. Hierdurch können auch FTBFS-Fehler (FTBFS = Failed To Build From Source) vermieden werden. Unter Umständen kann es jedoch noch zu solchen Fehlermeldungen kommen.

16.4.2 Konfiguration des `Pbuilders`

Zunächst muss das Verzeichnis */var/cache/pbuilder/result* angelegt werden. Dieses Verzeichnis muss für den Nutzer beschreibbar sein. Danach wird die Konfiguration des *pbuilder* durchgeführt.

Die Konfiguration des *pbuilder* ist nicht völlig trivial, was gegebenenfalls die Fehlersuche erschwert. Daher gehen wir hier sehr ausführlich darauf ein.

Die Standard-Konfiguration wird aus der Datei */usr/share/pbuilder/pbuilderrc* genommen.

```
 # pbuilder defaults; edit /etc/pbuilderrc to override these and see # pbuilderrc.5 for documentation

# Set how much output you want from pbuilder, valid values are # E => errors only # W => errors and warnings # I => err

BASETGZ=/var/cache/pbuilder/base.tgz #EXTRAPACKAGES="" #export DEBIAN_BUILDARCH=athlon BUILDPLACE=/var/cache/pbuilder/b
```

```
# specifying the distribution forces the distribution on "pbuilder update" #DISTRIBUTION=sid # specifying the architectur
# make debconf not interact with user export DEBIAN_FRONTEND="noninteractive"
#for pbuilder debuild BUILDSOURCEROOTCMD="fakeroot" PBUILDERROOTCMD="sudo -E" # use cowbuilder for pdebuild #PDEBUILD_PBU
# Whether to generate an additional .changes file for a source-only upload, # whilst still producing a full .changes file
# additional build results to copy out of the package build area #ADDITIONAL_BUILDRESULTS=(xunit.xml .coverage)
# command to satisfy build-dependencies; the default is an internal shell # implementation which is relatively slow; ther
# Arguments for $PBUILDERSATISFYDEPENDSCMD. # PBUILDERSATISFYDEPENDSOPT=()
# You can optionally make pbuilder accept untrusted repositories by setting # this option to yes, but this may allow remo
# Option to pass to apt-get always. export APTGETOPT=() # Option to pass to aptitude always. export APTITUDEOPT=()
# Whether to use debdelta or not. If "yes" debdelta will be installed in the # chroot DEBDELTA=no
#Command-line option passed on to dpkg-buildpackage. #DEBBUILDOPTS="-IXXX -iXXX" DEBBUILDOPTS=""
#APT configuration files directory APTCONFDIR=""
# the username and ID used by pbuilder, inside chroot. Needs fakeroot, really BUILDUSERID=1234 BUILDUSERNAME=pbuilder
# BINDMOUNTS is a space separated list of things to mount # inside the chroot. BINDMOUNTS=""
# Set the debootstrap variant to 'buildd' type. DEBOOTSTRAPOPTS=( '--variant=buildd' '--force-check-gpg' ) # or unset it
# Keyrings to use for package verification with apt, not used for debootstrap # (use DEBOOTSTRAPOPTS). By default the deb
# Set the PATH I am going to use inside pbuilder: default is # "/usr/sbin:/usr/bin:/sbin:/bin" export PATH="/usr/sbin:/us
# SHELL variable is used inside pbuilder by commands like 'su'; # and they need sane values export SHELL=/bin/bash
# The name of debootstrap command, you might want "cdebootstrap". DEBOOTSTRAP="debootstrap"
# default file extension for pkgname-logfile PKGNAME_LOGFILE_EXTENSION="_$(dpkg --print-architecture).build"
# default PKGNAME_LOGFILE PKGNAME_LOGFILE=""
# default AUTOCLEANAPTCACHE AUTOCLEANAPTCACHE=""
#default COMPRESSPROG COMPRESSPROG="gzip"
# pbuilder copies some configuration files (like /etc/hosts or # /etc/hostname) # from the host system into the chroot. I
```

Diese Werte können gegebenenfalls von Werten in der Datei */etc/pbuilderrc* überschrieben werden.

Nach einer Neuinstallation sieht die */etc/pbuilderrc* wie folgt aus:

```
# this is your configuration file for pbuilder. # the file in /usr/share/pbuilder/pbuilderrc is the default template. #
```

Diese habe ich wie folgt eingerichtet:

```
# this is your configuration file for pbuilder. # the file in /usr/share/pbuilder/pbuilderrc is the default template. #
```

Neben der globalen Konfigurationsdatei */etc/puilderrc* kann auch eine nutzerspezifische Datei *~/.pbuilderrc* angelegt werden. Der Inhalt dieser Datei überschreibt die systemweiten Einstellungen.

```
# BINDMOUNTS is a space separated list of things to mount # inside the chroot. BINDMOUNTS="/var/local/repository" # lo

# Added after reading: # https://lists.debian.org/debian-backports/2018/09/msg00021.html

# List of Debian suites. DEBIAN_SUITES=($UNSTABLE_CODENAME $TESTING_CODENAME \$STABLE_CODENAME $STABLE_BACKPORTS_SUITE

# Mirrors to use. Update these to your preferred mirror. DEBIAN_MIRROR="ftp.de.debian.org"

# Added after reading https://wiki.debian.org/cowbuilder BASEPATH="/var/cache/pbuilder/base.cow/"
```

Neben den systemweiten und nutzerspezifischen Konfigurationen werden auch paketspezifische Konfigurationen des Pbuilders benötigt. Diese Dateien können im Projekt-Verzeichnis abgelegt und mit *--configfile <Konfigurationsdatei>* eingelesen werden. Mit dieser Konfiguration werden eventuell schon vorhandene Werte überschrieben. Hier können dann die Informationen zu den Projekten abgelegt werden, wenn für diese Veröffentlichungen im Backports-Zweig benötigt werden. Diese Datei wird anch allen anderen Kovigurationsdateien eingelesen.

Siehe dazu auf Seite
`https://wiki.debian.org/BuildingFormalBackports`
den Abschnitt *#Advanced:_Building_multi-dependency_packages*

Dies kann auch in der Datei *~/.pbuilderrc* hinzugefügt werden, wenn es diese Datei gibt. Als MIRROR wird ein gängiger gut erreichbarer `Debian`-Mirror angegeben. Falls vorhanden, kann hier auch die Adresse eines vorhandenen lokalen Spiegel eingetragen werden.

Für die *Hook*-Skripte ist ein Verzeichnis *~/.pbuilder/* anzulegen.

16.4.3 Hooks einrichten

Hooks sind Skripte, die an bestimmten, vordefinierten Punkten während des Bauprozesses Dinge erledigen. Mit den sogenannten *Hooks* (Haken) kann der Prozess in der Build-Chroot auch an vordefinierten Positionen unterbrochen werden, um noch manuell in den Prozess eingreifen zu können.

Die *Hook*-Skripte befinden sich im Verzeichnis *~/.pbuilder/*

Der Name des Hook-Scriptes bestimmt, an welcher Stelle im Bauprozess der *Hook* ausgeführt wird.

Dabei gilt folgende Konversation:

```
X<digit><digit><whatever-else-you-want-as-name>
```

Das "‘X"’ (A bis G) bestimmt die Hook-Klasse, die folgenden 2 Ziffern die Reihenfolge, in der die Hooks einer Klasse ausgeführt werden. Der Rest dient als Beschreibung.

Leider entspricht die Reihenfolge, in der die Klassen im Build-Prozess ausgeführt werden, nicht der alphabetischen Reihenfolge.

A Ist für das –build Ziel. Es wird vor dem Baubeginn ausgeführt. D.h. nach dem Entpacken des Bausystems, des Quelltextes und nach der Erfüllung der Bau-Abhängigkeit.

B Wird ausgeführt, nachdem das Bausystem den Bau erfolgreich abgeschlossen hat, bevor das Bauergebnis zurückkopiert wird. - Unterbrechung nach erfolgreichem Bauen

C Wird nach einem Build-Fehler, vor der Bereinigung ausgeführt. - Unterbrechung nach gescheitertem Build

D Wird vor dem Entpacken der Quelle innerhalb der chroot-Umgebung ausgeführt, nachdem die chroot-Umgebung eingerichtet wurde. Erstellt $TMP und $TMPDIR wenn erforderlich. Dies wird aufgerufen, bevor die Build-Abhängigkeit befriedigt ist. Auch nützlich für den Aufruf von apt update. - Möglichkeit zum Editieren der Sources.list.

E Wird ausgeführt, nachdem *pbuilder –update* und *pbuilder –create* die Arbeit von apt-get mit dem chroot beendet hat, bevor das Kernel-Dateisystem (/proc) umountet und der Tarball aus dem chroot erzeugt wird.

F Is executed just before user logs in, or program starts executing, after chroot is created in –login or –execute target.

G Is executed just after debootstrap finishes, and configuration is loaded, and pbuilder starts mounting /proc and invoking apt install in –create target.

H Wird unmittelbar nach dem Auspacken von chroot, mounting proc und jedem in BINDMOUNTS angegebenen bind mount ausgeführt. Es wird für jedes Ziel ausgeführt, das die entpackte chroot benötigt. Es ist nützlich, wenn Sie die chroot-Einbauten dynamisch ändern wollen, bevor irgendetwas anfängt, sie zu benutzen.

I Wird ausgeführt, nachdem das Bausystem den Bau erfolgreich a bgeschlossen hat, nach dem Zurückkopieren der Build-Ergebnisse.

16.4.4 Hooks - Beispiele

Diese stehen alle im Verzeichnis: *~/.pbuilder/*

16.4.4.1 Hook A

Dieser hook könnte z.B. *A10shell* heißen.

64 ⟨*Hook-A* 64⟩≡

```
#!/bin/bash # example file to be used with --hookdir
invoke shell before build starts.

BUILDDIR="${BUILDDIR:-/tmp/buildd}"

apt-get install -y "${APTGETOPT[@]}" nano less cd "$BUILDDIR"/*/debian/.. echo "Hook A -the depe
are installed. Now the build can start." echo "Please use CTRL-D to continue" /bin/bash </dev/tt
2>/dev/tty
```

16.4.4.2 Hook B

Dieser hook könnte z.B. *B20shell* heißen.

65a ⟨*Hook-B* 65a⟩≡
```
#!/bin/bash # example file to be used with --hookdi
invoke shell if build fails.

BUILDDIR="${BUILDDIR:-/tmp/buildd}"

# apt-get install -y "${APTGETOPT[@]}" vim less cd "$BUILDDIR"/*/debian/.. echo "Hook B -The b
built successfully" echo "You can check it with ls -la ../" /bin/bash </dev/tty >/dev/tty 2>/d
```

16.4.4.3 Hook C

Hier können dann noch Pakete hinzu installiert werden, die zwingend benötigt werden, wenn der Build fehlschlägt.

Dieser hook könnte z.B. *C10shell* heißen.

65b ⟨*Hook-C* 65b⟩≡
```
#!/bin/bash # example file to be used with --hookdi
invoke shell if build fails.

BUILDDIR="${BUILDDIR:-/tmp/buildd}"

apt-get install -y "${APTGETOPT[@]}" vim less mc unzip locate cd "$BUILDDIR"/*/debian/.. echo
-The build wasn't built successfully" echo "After analysing the errors you can continue with u
/bin/bash </dev/tty >/dev/tty 2>/dev/tty
```

16.4.4.4 Hook D

65c ⟨*Hook-D* 65c⟩≡
```
#!/bin/bash # example file to be used with --hookdi
invoke shell before unpacking the source # inside the chroot

BUILDDIR="${BUILDDIR:-/tmp/buildd}"

apt-get install -y "${APTGETOPT[@]}" less nano #cd "$BUILDDIR"/*/debian/.. echo "Hook D -" ech
unpacking the sources the dependencies" echo "can be downloaded and unpacked." echo "Please us
to continue" /bin/bash </dev/tty >/dev/tty 2>/dev/tty
```

16.4.4.5 Hook E

65d ⟨*Hook-E* 65d⟩≡
```
echo "Hook E" echo "Please use CTRL-D to cont
/bin/bash </dev/tty >/dev/tty 2>/dev/tty
```

16.4.4.6 Hook F

65e ⟨*Hook-F* 65e⟩≡
```
echo "Hook F" echo "Please use CTRL-D to cont
/bin/bash </dev/tty >/dev/tty 2>/dev/tty
```

16.4.4.7 Hook G

66a ⟨*Hook-G* 66a⟩≡
```
echo "Hook G" echo "Please use CTRL-D to conti
/bin/bash </dev/tty >/dev/tty 2>/dev/tty
```

16.4.4.8 Hook H

66b ⟨*Hook-H* 66b⟩≡
```
#!/bin/bash # example file to be used with --hookdir
invoke shell if build fails.

BUILDDIR="${BUILDDIR:-/tmp/buildd}"

echo "Hook H" echo "Executed after preparing the chroot \n and before installing the depencencie
"Here you can include dependency from e.g a local repo for testing." echo "Next the source code
package to be built is unpacked." echo "Please use CTRL-D to continue" /bin/bash </dev/tty >/dev
2>/dev/tty
```

16.4.4.9 Hook I

66c ⟨*Hook-I* 66c⟩≡
```
#!/bin/bash # example file to be used with --hookdir
invoke shell if build fails.

BUILDDIR="${BUILDDIR:-/tmp/buildd}"

#apt-get install -y "${APTGETOPT[@]}" vim less #cd "$BUILDDIR"/*/debian/.. echo "Hook I" echo "P
use CTRL-D to continue" /bin/bash </dev/tty >/dev/tty 2>/dev/tty
```

16.4.5 Alternative *chroot*-Umgebungen

Es hat sich bewährt, für die verschiedenen `Debian`-Zweige eigene *chroot*-Umgebungen bereitzustellen.

Dazu wird eine Kopie von dem Verzeichnis */var/cache/pbuilder/base.cow* angelegt. Darin kann dann z.B. Die Datei */etc/apt/sources.lists* entsprechend angepasst werden.

Dabei kann es bei einer Aktualisierung der Pakete notwendig sein, dass zunächst auch Abhängigkeiten dieser Pakete aktualisiert werden müssen. Diese stehen jedoch nur mit Zeitverzögerung im Repo zur Verfügung. Hier kann es helfen für die weiteren Tests schon einmal die Zeile

```
deb http://incoming.debian.org/debian-buildd buildd-unstable main
```

in der */etc/apt/sources.lists* zu aktivieren. Damit stehen die dortigen Pakete für das Bauen in der *pbuilder*-Chroot zur Verfügung.

16.5 Weitere Chroot-Systeme

Neben dem Bauen in einer *pbuilder*-Chroot gibt es noch weitere Situationen, in denen die Nutzung eines separates Systems nützlich sein kann. Dies gilt besonders für das Ausführen von *mh_make*. (Kapitel **??**, Seite **??**)

Diese Einrichtung einer *Maven-Chroot* wird als Beispiel beschrieben:

Zunächst ist das Paket *debootstrap*, sofern noch nicht vorhanden, zu installieren. Es wird mit

```
sudo mkdir /srv/maven-chroot
```

ein entsprechendes Verzeichnis angelegt. Die Chroot selber wird mit

```
sudo /usr/sbin/debootstrap --arch amd64 sid \/srv/maven-chroot http://ftp.de.debian.org
```

erstellt.[32]

Danach kann der Nutzer *root* mit dem Befehl *chroot* ein neues Wurzelverzeichnis starten.

Die konkreten Befehle lauten (als root):

```
# mount -o bind /proc /srv/maven-chroot/proc # mount devpts /dev/pts -t devpts # LANG=C
```

Die letzte Zeile startet die angelegte Chroot.

Dieser User *root* kann dann nicht mehr auf Dateien außerhalb des neuen Wurzelverzeichnisses zugreifen.

Die Chroot-Umgebung kann wie folgt wieder beseitigt werden:

```
sudo umount /srv/maven-chroot/proc # Unmount first! sudo rm -rf /srv/maven-chroot/
```

16.6 Quilt fürs Patchen einrichten

Mit diesem Skript wird für das Patchen unter anderem *dquilt* verwendet. Dies ist eine debian-spezifische Anpassung für *quilt* .

Um diese Anpassung zu erzeugen, bedarf es der Datei *.quiltrc-dpkg* mit folgendem Inhalt [1]:

68 ⟨*DQuilt* 68⟩≡
```
d=. ; while [ ! -d $d/debian -a ‘readlink -ev $d‘!= / ]; do d=$d/..; done if
$d/debian ] &&[ -z $QUILT_PATCHES ]; then # falls in Debian-Paketbaum mit ungesetztem $QUILT_PAT
QUILT_PATCHES="debian/patches" QUILT_PATCH_OPTS="--reject-format=unified" QUILT_DIFF_ARGS="-p ab
--no-index --color=auto" QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index" QUILT_COLORS="dif
31:diff_hunk=1;33:diff_ctx=35:diff_cctx=33" if ! [ -d $d/debian/patches ]; then mkdir $d/debian/
fi fi
```

Dazu gehört für den manuellen Betrieb auch der Eintrag in der Datei textit~/.bashrc . Dieser Eintag sieht wie folgt aus:

```
# wird fuer die Patch-Verfolgung mit Quilt benoetigt alias dquilt="quilt --quiltrc=${HOM
```

Zu beachten ist, dass Einstellungen in der Datei *~/.bashrc* bei der Ausführung dieses Programmes nicht beachtet werden. Es müssen also alle notwendigen Einstellungen im Skript komplett abgebildet werden.

Die Verwendung von *quilt* bzw. *dquilt* wird in Kapitel *Nutzung von Quilt* (Kapitel **??**, Seite **??**) beschrieben.

[1] https://www.debian.org/doc/manuals/maint-guide/modify.html

Chapter 17

Git einrichten

17.1 Branches

Das Git-Repositorium eines Debian-Paketes hat in der Regel mindestens folgende Zweige:

- debian/sid
- upstream
- pristine-tar

Der Branch *debian/sid* kann auch *master* genannt werden.

Es kann einen zusätzlichen Branch für *experimental* geben. Außerdem können Branches für *backports* und *update-proposal* angelegt werden.

Solche weiteren Branches können auch vom Programm-Skript angelegt werden. (Kapitel **??**, Seite **??**)

17.2 Mergen

Wird von *debian/experimental* nach *debian/sid* gemergt, wird in der Regel ein Fast-Forward-Merge durchgeführt.

Dies ist dann der Fall, wenn - wie hier - zunächst in debian/experimental weitere Aktualisierungen zum Testen eingepflegt wurden, dann aber die Entwicklung in *debian/sid* fortgeführt werden soll.

Wurden jedoch in der Zwischenzeit auch im Branch *debian/sid* Änderungen hinzugefügt, kann nur ein Recursive-Merge durchgeführt werden.

Gegebenenfalls müssen einige Anpassungen, z.B. im Verzeichnis *debian/* einzeln ausgewählt und dem jeweiligen Zweig hinzugefügt werden (*git cherry-pick*).

Eine dazu bewährte Befehlszeile ist:

```
git cherry-pick --edit -x <commit>
```

Sollen mehrere Commits auf einmal dem Zweig zugeführt werden, gilt folgende Befehlszeile:

```
git cherry-pick --no-commit <commit> <commit> \dots
```

17.3 *gbp.conf*

Das Programmskript nutzt die Anwendungen aus dem `Debian`-Paket *git-buildpackage*.

git-buildpackage (gbp) kann und sollte konfiguriert werden.

Die Konfigurationsdatei *gbp.conf* wird zur Steuerung dieser Anwendung verwandt. Sie kann an verschiedenen Stellen im Dateisystem platziert werden.

17.3.1 Reihenfolge

Die Konfigurationsdateien für gbp werden in folgender Reihenfolge eingelesen:

1. /etc/git-buildpackage/gbp.conf, die systemweite Konfigurationsdatei
2. ~/.gbp.conf, die nutzerspezifische Konfigurationsdatei
3. debian/gbp.conf, Konfiguration für das Repositorium oder den Branch
4. .git/gbp.conf, Konfiguration für das lokale Repositorium

Alle Konfigurationsdateien haben das gleiche Format. [1]

Durch Setzen der Umgebungsvariablen *GBP_CONF_FILES* kann diese Reihenfolge überschrieben werden. Der Inhalt dieser Variable kann mit *echo $GBP_CONF_FILES* ermittelt werden. [2].

17.3.2 Abschnitte in der *gbp.conf*

Es gibt verschiedene Abschnitte in der *gbp.conf*. Diese Abschnitte sind alle optional.

Für jeden *gbp*-Befehl[3] kann ein eigener Abschnitt erstellt werden. Zusätzlich gibt es einen Abschnitt, der für alle Befehle gilt

Einige wichtige Abschnitte werden im Folgenden aufgeführt.

[DEFAULT] In diesem Abschnitt angegebenen Optionen werden auf alle *gbp*-Befehle angewandt. [4]

[import-orig] Die Optionen dieses Abschnittes überschreiben die des Abschnittes *[DEFAULT]*. Sie werden auf den Befehl *gbp import-orig* angewandt.

[pq] Die Optionen dieses Abschnittes werden auf den Befehl *gbp pq* angewandt und überschreiben die des Abschnittes *[DEFAULT]*.

[dch] Die Optionen dieses Abschnittes werden auf den Befehl *gbp dch* angewandt und überschreiben die Optionen des Abschnittes *[DEFAULT]*.

[buildpackage] Die Optionen dieses Abschnittes werden auf den Befehl *gbp buildpackage* angewandt und überschreiben die Optionen des Abschnittes *[DEFAULT]*.

---

[1] [3], Abschnitt *Configuration Files* und die Manpage zu *gbp-conf*

[2] [3] Abschnitt *Configuration Files/Overriding Parsing Order*

[3] [3] Manpage zu *gbp-conf*

[4] [3] Manpage zu *gbp-conf*

17.3.3 Syntax der Optionen

Die Optionen in den Abschnitten der *gbp.conf* werden aus den Befehlszeilenoptionen gebildet. Die möglichen Optionen zu den einzelnen *gbp*-Befehlen können der jeweiligen Manpage entnommen werden.

Diese werden ohne das einleitende doppelte Minuszeichen angegeben. Aus *--patch-num-format=%02d_* als Befehlszeilenoption wird beispielsweise *patch-num-format=%02d_*.

Im Falle von *gbp buildpackage* ist zusätzlich auch *git-* wegzulassen[5].

Der Eintrag *pbuilder-options=PBUILDER_OPTION* in der *gbp.conf* entspricht also *--git-pbuilder-options=PBUILDER_OPTION*.

17.3.4 Beispiel

Im Homeverzeichnis des Nutzers kann eine Datei *~/.gbp.conf* beispielsweise wie folgt angelegt werden:

71 ⟨*gbp.conf* 71⟩≡

```
[DEFAULT] sign-tags = True # keyid for signing the package keyid = 0x<keyid>pristine-tar = Tru
you want to use normally sbuild # builder = sbuild

[buildpackage] postbuild = lintian $GBP_CHANGES_FILE cleaner = /bin/true # If you want to use
pbuilder # pbuilder = True pbuilder-options = --source-only-changes --hookdir /home/mechtilde/

#[buildpackage] # use a build area relative to the git repository # export-dir=../build-area #
the same build area for all packages use an absolute path: #export-dir=/home/debian-packages/b

[dch] id-length = 7

# Options only affecting gbp pq [pq] #patch-numbers = False # The format specifier for patch n
#patch-num-format = ' patch-num-format = ' # Whether to renumber patches when exporting patch
#renumber = False renumber = True # Whether to drop patch queue after export #drop = False
```

[5] Manpage zu *gbp-conf*[3]

In der Datei */etc/git-buildpackage/gbp.conf* werden die Konfigurationsmöglichkeitenaufgeführt.

17.4 Git-Repositorien auf eigener Infrastruktur

17.4.1 Lokales Git-Repositorium

Das lokale Git-Repositorium wird entweder vom Skript angelegt (Kapitel **??**, Seite **??**) oder durch Klonen erzeugt (Kapitel **??**, Seite **??**)

Ihm können weitere Branches hinzugefügt werden (Kapitel **??**, Seite **??**).

17.4.2 Eigener Git-Server

Das lokale Git-Repositorium kann auch auf einem eigenen Git-Server "'gespiegelt"' werden.

Die Einrichtung des Servers erfolgt manuell. Die Verwendung von *cgit* odr*gitweb* erleichtern den Zugriff.

Im Programm-Skript können der Name oder die IP des eigenen Git-Servers eingegeben werden (Kapitel **??**, Seite **??**).

Der "'Workflow"' ist dann wie folgt: Nach der Anlage der Konfigurationsdatei wird zunächst der Name oder die IP des eigenen Git-Servers eingegeben (Kapitel **??**, Seite **??**), bevor mit dem Bauen eines neuen Paketes begonnen wird (Kapitel **??**, Seite **??**).

Das "'fertige"' Paket kann dann auf den eigenen Git-Server hochgeladen werden (Kapitel **??**, Seite **??**).

17.5 Salsa-Repositorien

Ein Git-Repositorium auf *salsa.debian.org* wird oft manuell eingerichtet.

Nach dem Login und der Authentifizierung auf *https://salsa.debian.org* kann dort im jeweiligen Team ein neues Repositorium angelegt werden.

Für die Anlage von neuen Projekten auf `https://salsa.debian.org` sind entsprechende Rechte erforderlich. Dies können die Rechte eines `Debian Developers` sein. Für die team-betreuten Projekte können diese Rechte auch an weitere Personen von den Betreuenden vergeben werden.

Ein Beschreibung der Anlage eines Projektes im Java-Team erfolgt in Kapitel 17.5.2, Seite **??**.

17.5.1 Anlage eines Salsa-Repositoriums

Nach dem Login auf der Seite `https://salsa.debian.org` und der Auswahl des passenden Projektes für das neue Paket, wird mit dem Klick auf den Button *Neues Projekt* die entsprechende Seite aufgerufen. Dort gibt man den Projektnamen ein. Dies ist meist der Name des Quellcodepaketes. Als Sichtbarkeitsgrad (Visibility Level) wird *Öffentlich* ausgewählt. Dann folgt der Button *Create project*.

Auf der folgenden Web-Seite gibt es noch einige Erläuterungen. Vieles davon wird zunächst lokal mit dem beschriebenen Programm angelegt.

In der linken Navigationsleiste werden nun im Bereich *Einstellungen* weitere Konfigurationen zum Projekt vorgenommen.

Hier ist es wichtig unter *CI/CD* den Pfad und den verwendeten Dateinamen anzugeben.

CI steht für Continuous Integration

CD steht für Continuous Delivery

Die hier verwendete Datei heißt *salsa-ci.yml* (Kapitel **??**, Seite **??**) im Verzeichnis *debian/*.

Das Build-Skript trägt das Salsa-Repositorium als "‘Remote-Repositorium"’ ein und erinnert den Nutzer an die Anlage auf *salsa.debian.org* (Kapitel **??**, Seite **??**)

17.5.2 `Java`-Team

Salsa-Repositorien, die einem speziellen Projekt (wie beispielsweise dem `Java`-Team) zugeordnet sind, sollten möglichst einheitlich angelegt werden. Häufig wird hierzu ein Skript vom Projekt bereitgestellt, welches hierzu verwandt werden soll.

Dazu wechselt man in das gewünschte Team-Verzeichnis.

Figure 17.1: Information zum `Java`-Team.

17.5.2.1 Quelle des Skripts

Unter `https://salsa.debian.org/java-team/pkg-java-scripts/blob/master/setup-salsa-repository` kann das Skript des *Java-Teams* heruntergeladen werden. Es ist auch im Anhang beigefügt (Kapitel **??**, Seite **??**).

17.5.2.2 Abhängigkeiten

Dieses *Setup-Skript* nutzt *jq*. *jq* ist ein leichtgewichtiger und flexibler *JSON*-Prozessor für die Kommandozeile. Er hat nur minimale Laufzeitabhängigkeiten. Es gibt ein gleichnamiges `Debian`-Paket. Dieses Paket muss lokal installiert sein (Kapitel 16.1.2, Seite 58).

17.5.2.3 Zugangstoken beschaffen

Für die Anpassung wird noch ein projekt-spezifischer Token benötigt.

Nach dem Login auf *salsa.debian.org* wird im Dropdown-Menü des Nutzers *Einstellungen* gewählt. Damit wird die Seite *Benutzereinstellungen* des eingeloggten Nutzers aufgerufen.

In der linken Leiste befindet sich nun der Eintrag *Zugangs-Token*. Dort wird die Seite *https://salsa.debian.org/profile/personal_access_tokens* zur Erstellung eines solchen Token aufgerufen. Er wird für jedes Projekt neu generiert.

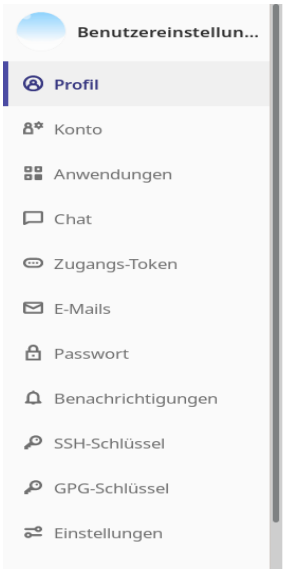

Figure 17.2: Zugangstoken erstellen

Dort wird der Name des zu erstellenden Projektes eingegeben. Ferner wird ein Ablaufdatum für das Token eingegeben. Danach wird der Gültigkeitsbereich des Tokens festgelegt. Als *Scope* ist hier *api* auszuwählen. Mit dem Klick auf den Button *Create personal access token* erscheint oben auf der Seite der Zugangstoken.

17.5.2.4 Token eintragen

Das generierte Token ist als *SALSA_TOKEN* in das Skript einzutragen. Das Kommentarzeichen ist zu entfernen

Es erscheint sinnvoll, dieses Skript jeweils im Projektverzeichnis abzulegen.

17.5.2.5 Skript aufrufen

Das Skript wird nun mit dem Namen des neuen Projektes (Paketes) als Parameter aufgerufen:

```
./setup-salsa-repository.sh <packagename>
```

Danach werden folgende Meldungen ausgegeben (anhand des Beispiels BeanValidationApi):

```
./setup-salsa-repository.sh beanvalidation-api Creating the beanvalidation-api reposito

Done! The repository is located at https://salsa.debian.org/java-team/beanvalidation-api
```

Erscheint lediglich die erste Zeile, sind das verwendete Skript (s. Kapitel **??**, Seite **??**) und Token zu prüfen.

17.6 Aufgaben auf *salsa.debian.org*

17.6.1 Merge Request

Manchmal gibt es auch sogenannte Merge Request, in dem andere Patches bereitstellen.

Auf *salsa.debina.org* gibt es in der linken Navigationsleiste dann einen Eintrag "‘Merge-Requests"’. Dort kann dieser dann bearbeitet werden.

Dies erfolgt durch Klicken auf die Commit-Message

Chapter 18

Paketieren jenseits vom Zweig *Unstable*

Es gibt verschiedene Gründe, warum von dem generellen Weg, dass neue Upstream-Versionen ausschließlich für *Unstable = sid* paketiert werden, abgewichen werden kann und wird.

Ein wesentlicher Grund ist das Vorhandensein eines schwerwiegenden Fehlers oder eines Sicherheitsproblems. Schwerwiegende Fehler in einem Paket werden in der Regel durch einen entsprechend eingestuften Fehlerbericht gemeldet.

Eine weitere Abweichung von dieser Regelung gibt es bezüglich der Pakete der `Mozilla`-Suite, `Firefox` und `Thunderbird`. Diese Pakete werden zeitnah nach ihrem Upstream-Release für die *ESR = Extended Support Release* Version auch als *Security-Updates* (Kapitel 18.1, Seite 78) für das *Stable*-Release bereitgestellt.

Daraus können sich z.B. für die Erweiterungen, hier als Webextension (Kapitel 12, Seite 49) beschrieben, Inkompatibilitäten mit der dann aktuellen Version im *Stable*-Release ergeben. Dies ist ebenfalls ein Grund eine Aktualisierung im *Stable*-Release vorzunehmen (s.a. Kapitel 18.2, Seite 78).

Daneben ist es bei Desktop-Anwendungen oft wünschenswert, aktuellere Versionen derselben in einem *Stable*-Release zu nutzen. Dazu werden die gewünschten Pakete als sogenannte *Backports* (d.h. Rückportierungen) zur Verfügung gestellt. (s. Kapitel 18.3, Seite 80)

Es gibt auch gute Gründe, Pakete zunächst nach *Experimental* hochzuladen. Dies gilt besonders für neue Pakete. In dem Zeitraum, in dem die bestehenden Versionen "'eingefroren"' sind und nur Fehlerbehebungen für das nächste Release erlaubt sind, werden neue Versionen oftmals schon nach experimental hochgeladen. (s. Kapitel 18.5, Seite 80)

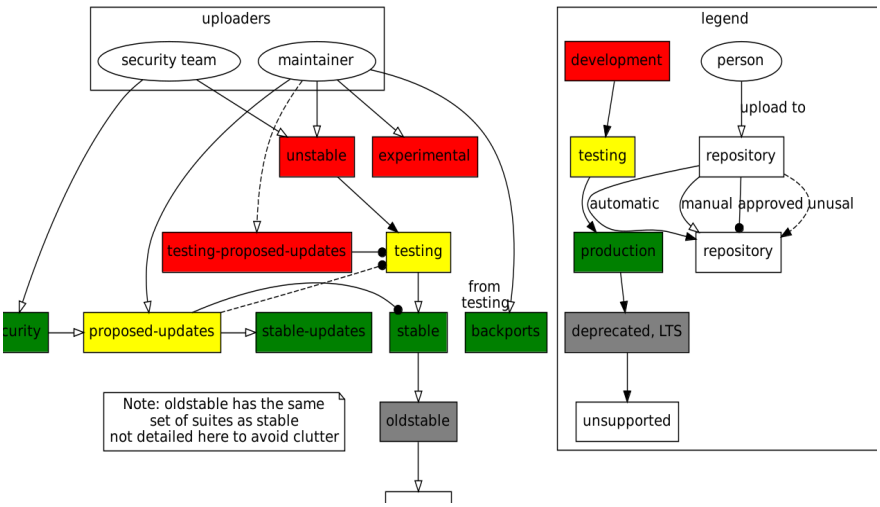

Figure 18.1: Arbeitsabläufe [33]

18.1 Security-Updates

Wichtig ist die Rückportierung von Änderungen an Quellcode-Dateien wegen Sicherheitsproblemen.

Immer, wenn ein Sicherheitsproblem bekannt wird, soll der Maintainer mit dem Sicherheits-Team zusammenarbeiten, um für das *Stable*-bzw. *Oldstable*-Release eine korrigierte Version bereitzustellen. Die Fehlerbehebung sollte gezielt erfolgen. Weitere Inforamtionen befinden sich in den Kapiteln 3.1.2 und 5.8.5 der Debian-Entwicklerreferenz[9].

18.2 (Old-)Stable-Proposal

Sollen Pakete mit schwerwiegenden Fehlern, die nicht die IT-Sicherheit betreffen, aktualisiert werden, kann dies unter bestimmten Voraussetzungen in *Proposed-Updates* erfolgen.

Als Kriterien für das Hinzufügen von Paketen zu *stable-updates* sind Folgende genannt worden [1]:

- Die Aktualisierung ist dringend und nicht sicherheitsrelevant. Die Sicherheitsaktualisierungen erfolgen im oben genannten Wege (s. Kapitel 18.1, Seite 78).
- Das fragliche Paket ist ein Datenpaket, und die Daten müssen zeitnah aktualisiert werden (z.B. tzdata).
- Korrekturen an Paketen, die durch externe Änderungen beeinträchtigt sind und von denen kein anderes oder nur wenige andere Pakete abhängig sind.
- Pakete, die aktuell sein müssen, um nützlich zu sein (z.B. clamav).

[1] `https://lists.debian.org/debian-devel-announce/2011/03/msg00010.html`

Wer glaubt, dass die angetrebte Aktualisierung diese Kriterien erfüllt, sollte sich vertrauensvoll an das Release Team über die Mailingliste debian-release@lists.debian.org, und erläutern, dass diese Aktualisierung auch über *stable-updates* erfolgen sollte.

Das für die Rückportierung auf diesem Wege vorgesehene Paket sollte bereits in *Unstable* oder besser noch in *Testing* veröffentlicht worden sein.

Während Backports in einem eigenen Repositorium gesammelt werden, werden *Proposed-Updates* in das *Stable*-Proposed-Repositorium eingefügt. Diese Pakete werden dazu von den Debian-Entwicklern nach *Stable* hochgeladen. Dies gilt für Oldstable-Proposed-Updates entsprechend. Die Veröffentlichung erfolgt im nächsten Point-Release

18.2.1 Fehlerbericht

Eine notwendige Voraussetzung hierfür ist zunächst ein aussagekräftiger Fehlerbericht (Bug-Report).

Dieser Fehlerbericht benötigt folgende Parameter:

```
Package: release.debian.org Version: Severity: normal
```

Dieser Fehlerbericht wird gegen das Pseudopaket *release.debian.org* erstellt. Der Schweregrad eines solchen Fehlerberichtes ist im Allgemeinen maximal *normal*. Im Betreff wird der Name des Paketes aufgeführt, das für *Proposed-Updates* gebaut wird.

Diese *Proposed-Updates* werden wie dargelegt nur unter besonderen Umständen zugelassen. Das *Release-Team* entscheidet, ob die Veröffentlichung erfolgt.

Daher muss der Fehlerbericht eine Begründung enthalten, warum diese Version in *stable* aktualisiert werden soll. Sofern es bereits eine dazugehörige Fehlernummer gibt, ist diese anzugeben. Dieser Fehlerbericht **muss** einen Schweregrad von *important* oder höher aufweisen.

18.2.2 Anforderungen an einen Patch

Ein Fehler sollte mit einem möglichst kleinen und passgenauen Patch behoben werden können. Damit soll verhindert werden, dass neue Fehler entstehen. Es dürfen auch keine neuen Abhängigkeiten hinzukommen. Auch muss berücksichtigt werden, welche anderen Pakete von diesem Paket abhängen. [2]

Diese passgenauen Patches werden mit *debdiff <BisherigesPaket>.dsc <NeuesPaket>.dsc > <Dateiname>.txt* dokumentiert. Es wird die Differenz angegeben, die zwischen der aktuellen Paketversion in Testing/Sid (bisheriges Paket)und der momentan aktuelle Version (neues Paket) in Stable exisitiert.

[2] Kapitel 5.5.1 in der Developer-Reference[9]

18.2.3 Abhängigkeiten zu Mozilla-Paketen

Eine hinreichende Begründung für *Webextensions* kann auch die oben (s. Kapitel 18, Seite 77) erwähnte Ausnahmeregelung für die Pakete der Mozilla-Suite sein.

Für diese Erweiterungen können sich nämlich Inkompatibilitäten mit der dann aktuellen Version von Firefox oderThunderbird im *Stable*-Release ergeben. Dann sind die bisherigen Versionen unbrauchbar, was vor allem im produktiven Betrieb ein beachtliches Problem darstellt. (Kapitel ??, Seite ??)

18.3 Stable-Backports

Um Pakete für Backports bauen zu können, sind einige Vorbereitungen zu treffen.

Oft lässt sich ein solches Paket nicht ohne Weiteres in einer Stable-Umgebung bauen.

Dann werden weitere Pakete aus Backports benötigt. Pakete aus Backports werden aber nicht automatisch auch in einer solchen *Chroot* installiert.

Daher ist es notwendig, in */etc/apt/preferences* eine solche Installation zu ermöglichen.

Dazu werden im *pbuilder Hooks* verwendet (Kapitel 16.4.3, Seite 63).

https://wiki.debianforum.de/Pbuilder_-_personal_package_builder

18.4 Backports-Repositorium

Regelmäßig werden die Pakete für den Unstable-Zweig gebaut.Von dort gelangen sie dann - sofern keine Fehler gefunden werden - in den Testing-Zweig. Beim Release wird der Testing-Zweig zunächst eingefroren und dann zum neuen Stable-Zweig. Es wird dann ein neuer Testing-Zweig eröffnet.

Werden Programmversionen aus *testing* oder (ausnahmnsweise [3]) auch aus *unstable* in ein früheres Release übertragen, nennt man dies *backporting*.

Durch die an der Praxis orientierten Releasezyklen der stabilen Debian-Version ist es manchmal wünschenswert, Softwarepakete oder neuere Versionen aus Testing auch unter Stable verfügbar zu haben.

Hierzu dient das Backports-Repositorium.

Versionierung (z.B. +deb9u1)

Backports gibt es für das Stable- und Oldstable-Release.

18.5 Experimental

18.6 Backporten fremder Pakete

pristine-tar NMU Abhängigkeiten aus Backports

[3] in der Regel nur Sicherheits-Updates

Verweis auf (Kapitel **??**, Seite **??**) [4] für die Veröffentlichung nach *Proposed-Updates* für *Stable* bzw. *Old-Stable*.

18.7 Versionierung

Die Ursache bzw. Herkunft der Aktualisierung hat auch Einfluss auf die Versionierung. Dabei ist sicherzustellen, dass *dpkg* neuere Versionen korrekt als Upgrades interpretieren kann. Ein Blick auf die nächste zu erwartende Versionsnummer kann dabei helfen.

Beim Bauen für *experimental* wird folgender Versionseintrag in der Datei *debian/changelog* empfohlen:

<Versionsnummer>-<Revisionsnummer>~exp<Laufende Nummer>

Die Revisionsnummer ist in der Regel: 1

Beim Bauen für *Proposed-Updates* sind zwei Fälle zu unterscheiden.

Fall A Eine im *Stable*-Release vorhandene Version soll unter Beibehaltung dieser Versionsnummer korrigiert werden.

Fall B Ausnahmsweise soll eine neue Version in das Release aufgenommen werden.

Für den Versionseintrag in *debian/changelog* ist zwingend folgende Nomenklatur zu verwenden:

Fall A <Ursprüngliche Versions- und Revisionsnummer> +deb<Debian-Release-Nummer>u<Revisionsnummer des Updates>

Fall B <Versions- und Revisionsnummer aus Unstable/Testing>~deb<Debian-Release-Nummer>u<Revisionsnummer des Updates>

Die Verwendung der ~bewirkt, dass die Version in Stable kleiner als die in Testing (für das nächste Release) ist. Dies sichert den Aktualisierungspfad.

[4] [9] Abschnitt *upload-stable*

Chapter 19

Zum Start eine E-Mail

Bevor das Paketieren mit dem Ziel der Veröffentlichung im `Debian`-Repositorium beginnen kann, bedarf es einer E-Mail. Diese löst einen "'Bugreport"' aus, dessen Nummer in der Datei *debian/changelog* zu vermerken ist. Durch die Veröffentlichung des Paketes soll nämlich dann dieser Bugreport geschlossen werden.

Weitere Informationen dazu gibt es unter `https://www.debian.org/devel/wnpp/`

19.1 ITP - Intent To Package

ITP bedeutet "'Intent to Package"'. Dies bedeutet, dass an diesem Paket gearbeitet wird.

Dies ist die Bezeichnung eines Bug-Reportes, der gegen das Paket *WNPP* erstellt wird, was *Work-Needing and Prospective Package* bedeutet. Dies kann mit *Arbeitsbedürftige und voraussichtliche Pakete* übersetzt werden.

Dies sollte rechtzeitig angekündigt werden. Vielleicht gibt es ja noch Hinweise, was bei dem geplanten Paket zu beachten ist. Auch wird so verhindert, dass verschiedene Gruppen versuchen, dieses Paket für `Debian` zu paketieren.

Eine Möglichkeit ist, dies mit dem auf jedem Debian-System installierten Tool *reportbug* durchzuführen. Es gibt aber auch die Möglichkeit einfach eine E-Mail-Vorlage zu nutzen. Die folgende Vorlage basiert auf dem *itp_template* mit dem *reportbug* diese E-Mail erstellt.

```
 To: submit@bugs.debian.org Subject: ITP: <Source Name> - <Short Description> Package: w

* Package name : <Package Name> Version : x.y.z Upstream Author : Name <somebody@example

(Include the long description here.)

<And answer following questions:>

* Why is this package useful/relevant? * Is it a dependency for another package? * Do yo
```

Der Text in der ersten Zeile hinter dem *To:* kommt in die Adresszeile. Der Text in der zweiten Zeile hinter dem *Subject:* kommt in die Betreffzeile, wobei die Platzhalter durch den Namen des Quellcodes und einer kurzen Beschreibung ersetzt werden.

Wird das Paket hochgeladen, muss dies im Changelog verwerkt weren (Closes:#XXXXXX)

19.2 RFP - Request For Package

RFP bedeutet *Request for Package*. Dies bedeutet, dass ein solches Paket in `Debian` erwünscht ist.

Auch hierzu habe ich einmal eine E-Mail-Vorlage erstellt.

```
 To: submit@bugs.debian.org Subject: RFP: <Sourcecode Name> - <Short Description> Package

* Package name : <Package Name> Version : x.y.z Upstream Author : Name <somebody@example.

(Include the long description here.)

<And answer following questions:>

* Why is this package useful/relevant? Is it a dependency for another package? * Do you u
```

Auch hier gilt: Der Text in der ersten Zeile hinter dem *To:* kommt in die Adresszeile. Der Text in der zweiten Zeile hinter dem *Subject:* kommt in die Betreffzeile, wobei die Platzhalter durch den Namen des Quellcodes und einer kurzen Beschreibung ersetzt werden.

19.3 ITA - Intent To Adoption

Dies wird verwendet, wenn ein Paket, dass mit "‘O"’ oder "‘RFA"’ gekennzeichnet ist, übernommen werden soll.

Dazu muss der vorherige Fehlerbericht umbenannt und "‘O"’ bzw. "‘RFA"’ durch "‘ITA"’ ersetzt werden.

Dabei tragen Sie sich als Besitzer ein.

Soll ein Fehlerbericht umbenannt oder der Besitzer geändert werden, muss dies per E-Mail an *control@bugs.debian.org* oder direkt an den Fehlerbericht über die Nummer (xxxxxx@bugs.debian.org) erfolgen[1].

Dabei ist ein strukturierter *Pseudo-Header* zu nutzen. [2]

[1] `https://www.debian.org/devel/wnpp/`

[2] `https://www.debian.org/Bugs/Reporting#control`

19.4 RFA - Request for Adoption

19.5 RFH - Request For Help

19.6 O - Orphaned

19.7 RFS - Request For Sponsor

Wie auf `https://mentors.debian.net/sponsor/rfs-howto` beschrieben, wurde die Vorlage angepasst.

```
 To: submit@bugs.debian.org Subject: ITP: <Package Name> - <Short Description> Package:

Dear mentors,

I am looking for a sponsor for my package "<Source Name!:

* Package name : <Sour Name> Version : x.y.z Upstream Author : Name <somebody@example.or

It builds those binary packages:

<Name of the Binaries>

To access further information about this package, please visit the following URL:

https://mentors.debian.net/package/<package name>

Alternatively, one can download the package with dget using this command:

dget -x https://mentors.debian.net/debian/pool/main/<p>/<package name>/<package name>_x.

Changes since the last upload: [your most recent changelog entry]

Regards,
```

19.8 Änderungen am Fehlerbericht

Im Laufe eines solchen Prozesses kann es vorkommen, dass Änderungen am Fehlerbericht erfolgen müssen. Ein solche Änderung kann eine Änderung des Maintainers oder auch des Titels sein.

Dazu werden die Befehle des *Kontroll-E-Mail-Server* genutzt. Die Beschreibung dazu findet sich unter `https://www.debian.org/Bugs/server-control`

Für die Änderung des Titels wird dann zusätzlich eine Zeile wie folgt

```
 Control: retitle -1 <neuer Titel>
```

eingefügt.

Für die Änderung des Maintainers wird die folgende Zeile hinzugefügt:

```
Control: owner -1 <Neuer Maintainer oder wnpp@debian.org>
```

19.9 *usertags* hinzufügen

Im Laufe eines Maintainer-Lebens kommt es immer wieder vor, dass Fehlermeldungen mit tags versehen werden müssen oder sollten.

Z. B. ist es hilfreich für sogenannten *Bug Squashing Parties*, die geplanten und durchgeführten Fehlerbehebungen auch zu kennzeichnen.

Dafür wird eine E-Mail an den Fehlerbericht geschrieben. Adresse lautet dann *<Bugnummer>debian.org*. Gleichzeitig soll diese E-Mail auch *CC* an *controlbugs.debian.org* gehen.

Am Anfang einer solchen E-Mail steht dann:

```
user debian-release@lists.debian.org usertags -1 + <Titel der BSP> thank you
```

Chapter 20

Reportbug einrichten

Das Comand-Line-Interface ist bereits Bestandteil der Basisinstallation. Zusätzlich kann mit dem Paket *reportbug-gtk* noch ein graphisches Nutzerinterface.installiert werden.

Chapter 21

Autopkgtest

Chapter 22

Reproduzierbare Builds

22.1 Konfiguration von *sbuild*

Nach der Installation der benötigten Pakete:

```
sudo apt install sbuild schroot debootstrap apt-cacher-ng devscripts
```

wird die Konfigurationsdatei *sbuild* erstellt[34].

Dazu wird der folgende Abschnitt in die Datei ~/.sbuildrc kopiert. Diese Einstellungen ermöglichen es, lange Befehlszeilenoptionen für typische Arbeitsabläufe zu vermeiden und alle Tests nach dem Build auszuführen.

91a ⟨*.sbuild.rc* 91a⟩≡
```
############################################################################
PACKAGE BUILD RELATED (additionally produce _source.changes) ###############################
# -d $distribution = 'unstable'; # -A $build_arch_all = 1; # -s $build_source = 1; # --source-
(applicable for dput. irrelevant for dgit push-source). $source_only_changes = 1; # -v $verbos
```
⟨*.sbuild.rc5* 91b⟩

Passen Sie parallel=5 an die Ressourcen Ihres Systems an.

91b ⟨*.sbuild.rc5* 91b⟩≡ (91a)
```
# parallel build $ENV{'DEB_BUILD_OPTIO
'parallel=5'; ⟨.sbuild.rc6 91c⟩
```

Wenn stattdessen die Variable $run_lintian auf 0 gesetzet wird, wird die Ausführung von lintian deaktiviert.

Jede Post-Build-Paket-Testfunktion kann ausgeschaltet werden, indem die entsprechende Variable auf 0 gesetzt wird.

91c ⟨*.sbuild.rc6* 91c⟩≡ (91b)
```
############################################################################
# POST-BUILD RELATED (turn off functionality by setting variables to 0) ######################
$run_lintian = 1; $lintian_opts = ['-i', '-I']; ⟨.sbuild.rc7 91d⟩
```

Wenn die Variable $run_piuparts auf 0 gesetzet wird, wird die Ausführung von piuparts deaktiviert.

91d ⟨*.sbuild.rc7* 91d⟩≡ (91c)
```
$run_piuparts = 1; $piuparts_opts = ['--sch
'unstable-amd64-sbuild']; ⟨.sbuild.rc8 92⟩
```

Die Option *'--no-eatmydata'* für piuparts wird benötigt, wenn *schroot* mit "command-prefix=eatmydata" in */etc/schroot/chroot.d/unstable-amd64-sbuild-** konfigurieren. Dann muss es am Ende – nach einem Komma – noch hinzugefügt werden

Wenn Sie stattdessen die Variable $run_autopkgtest auf 0 setzen, wird die Ausführung von autopkgtest deaktiviert.

92 ⟨*.sbuild.rc8* 92⟩≡ (91d)

```
$run_autopkgtest = 1; $autopkgtest_root_args =
$autopkgtest_opts = [ '--', 'schroot', '

%r-%a-sbuild' ];
############################################################################## # PERL MAGIC ####
1;
```

Weitere Optionen finden Sie in der Manpage sbuild.conf.

Daneben muss der Nutzer mit

```
sudo sbuild-adduser \$LOGNAME
```

in der Gruppe *sbuild* eingerichtet werden. Dadurch wird Ihr Benutzername hinzugefügt, so dass dieser den Befehl *sbuild* verwenden kann.

Folgende Meldung erscheint nach der Einrichtung des Nutzers:

```
Now try a build:

cd /path/to/source sbuild-update -ud <distribution> (or "sbuild-apt <distribution> apt-ge
```

Dies bedeutet, dass *sbuild-adduser* die *sbuild*-Vorlagekonfiguration in /usr/share/doc/sbuild/examples/example.sbuildrc in die *~/.sbuildrc* jedes Benutzers kopiert, um sie als *sbuild*-Konfiguration für den jeweiligen Benutzer zu verwenden. Die *sbuild*-Einstellungen können hier angepasst werden. In der Regel sind aber keine Anpassungen notwendig. Wenn doch, sollten diese einmal pro Benutzer durchgeführt werden.

Die Erstellung der `Sbuild`-Chroot wird in Kapitel **??** (Seite **??**) beschrieben.

22.2 reprotest

Chapter 23

piuparts

Chapter 24

Schwierigkeiten überwinden

24.1 Ein Paket loseisen

Eine Schwierigkeit ergibt sich daraus, dass es vor einem geplanten Release eine Zeitspanne gibt, in der die Pakete nicht mehr automatisch von *unstable* nach *testing* migrieren.

Im vollständigen "'Freeze"' benötigen alle Pakete, die noch von *unstable* nach *testing* migrieren sollen, eine Entsperrung durch das Release-Team.[35]. Diese muss mit einem *Unblock Bug* beantragt werden.

24.1.1 Beantragung einer Entsperrung

Dazu wird zunächst eine Datei mit *debdiff* erstellt (s.a. Kapitel 18.2, Seite 78).

Damit wird die Differenz zwischen der Version in *testing* (alte Version) und *Unstable* (neue Version) mit der jeweiligen *dsc* erstellt.

Diese Differenz-Datei ist darauf zu prüfen, dass sie keine unwichtigen Änderungen für die gewünschte Fehlerbehebung hat.

Anschließend wird mit dem Werkzeug *reportbug* ein Fehlerbericht gegen das Paket *release.debian.org* erstellt und die Differenz-Datei angehängt. Dieser Fehlerbericht enthält eine ausführliche Begründung für die Änderungen und Verweise auf Fehlernummern. Ebenso enthält sie eine prägnante Beschreibung des Problems, das behoben wurde.

DAbei sollte auf folgende Fragen eingegangen werden.

```
 Package: release.debian.org User: release.debian.org@packages.debian.org Usertags: unbl

Please unblock package <source name>

(Please provide enough (but not too much) information to help the release team to judge

[ Reason ] (Explain what the reason for the unblock request is.)

[ Impact ] (What is the impact for the user if the unblock isn't granted?)
```

```
[ Tests ] (What automated or manual tests cover the affected code?)

[ Risks ] (Discussion of the risks involved. E.g. code is trivial or complex, key package

[ Checklist ] [ ] all changes are documented in the d/changelog [ ] I reviewed all change

[ Other info ] (Anything else the release team should know.)
```

24.2 Releasekritische Fehler beheben

Vor einer neue Veröffentlichung ist es oft notwenidg die Maintainer dabei zu untertützen, Fehler zu behegen, die einer Veröffentlichung des Paketes entgegenstehen.

Dies geschieht häufig auf dafür organisierten Verantaltungen [1].

Unter `https://www.debian.org/doc/manuals/developers-reference/pkgs.html#non-maintainer-uploads-nmus` ist das gewünschte Vorgehen beschrieben.

Wesentlich ist dabei, dass in der Datei *debian/changelog* (Kapitel **??**, Seite **??** ein entsprechender Eintrag in der zweiten Zeile erfolgt.

```
Non-maintainer upload
```

[1] `https://wiki.debian.org/BSP`

Part III

Anhang

List of Figures

Literaturverzeichnis

[1] Creative Commons. “Attribution-ShareAlike 4.0 International”. In: *https://creativecommons.org/* (Oct. 15, 2013). URL: https://creativecommons.org/licenses/by-sa/4.0/legalcode.

[2] Free Software Foundation, Inc. “GNU General Public License 3”. In: *https://www.gnu.org* (June 29, 2007). URL: https://www.gnu.org/licenses/gpl-3.0.de.html.

[3] Guido Günther. “Building Debian Packages with git-buildpackage”. In: *Git-Buildpackage* (2017). URL: https://honk.sigxcpu.org/projects/git-buildpackage/manual-html//.

[4] Norman Ramsey. “Noweb — A Simple, Extensible Tool for Literate Programming”. In: *NoWeb* (June 28, 2018). URL: https://www.cs.tufts.edu/%5Ctextasciitilde%20nr/noweb/.

[5] *Simple Packaging Tutorial*. Oct. 26, 2019. URL: https://wiki.debian.org/SimplePackagingTutorial.

[6] Debian Project. “Die Debian-Richtlinien für Freie Software (DFSG)”. German. In: *Debian-Gesellschaftsvertrag* (Apr. 26, 2004). URL: https://www.debian.org/social_contract.de.html.

[7] Ian Jackson, Christian Schwarz, and David A. Morris. “Debian Policy”. English. Version 4.6.1. In: *Debian Policy Manual* (May 22, 2022). Ed. by Die Debian-Policy-Gruppe. URL: https://www.debian.org/doc/debian-policy/. GNU General Public License Version 2+.

[8] Christopher Yeoh, Paul ’Rusty’ Russell, Daniel Quinlan. “Filesystem Hierarchy Standard”. English. Version 3.0. In: *Filesystem Hierarchy Standard* (Mar. 19, 2015). Ed. by The Linux Foundation LSB Workgroup. URL: https://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html. BSD.

[9] Ian Jackson et al. “Debian-Entwicklerreferenz”. Deutsch. Version 11.0.7. In: *Debian Entwicklerreferenz* (Oct. 24, 2020). Ed. by Hideki Nussbaum Lucas andYamane and Holger Levsen. URL: https://www.debian.org/doc/manuals/developers-reference/index.de.html. GNU General Public License Version 2+.

[10] Osamu Aoki. “Leitfaden für Debian-Betreuer. debmake-doc”. Deutsch. In: *Debmake-Doc* (Mar. 26, 2019). URL: https://www.debian.org/doc/devel-manuals#debmake-doc. Expat-Lizenz.

[11] Josip Rodin, Osamu Aoki. “Debian-Leitfaden für Neue Paketbetreuer. New Maintainer Guide”. Deutsch. Version 1.2.43. In: *Debian-Leitfaden für Neue Paketbetreuer* (Nov. 7, 2020). Ed. by Osamu Aoki. wird ersetzt durch Aoki, Leitfaden für Debian-Betreuer. URL: https://www.debian.org/doc/manuals/maint-guide/. GNU General Public License Version 2+.

[12] Axel Beckert and Frank Hofmann. “Debian-Paketmanagement”. In: *DPMB* (Feb. 7, 2021), p. 400. URL: https://book.dpmb.org/debian-paketmanagement.chunked/index.html.

[13] *ReproducibleBuilds*. Dec. 6, 2020. URL: https://reproducible-builds.org/.

[14] *The experimental repository*. Nov. 15, 2020. URL: https://wiki.debian.org/DebianExperimental.

[15] *The Debian GNU/Linux FAQ*. Aug. 12, 2019. URL: https://www.debian.org/doc/manuals/debian-faq/index.de.html.

[16] *How can i help*. Feb. 7, 2021. URL: https://wiki.debian.org/how-can-i-help.

[17] Steve Langasek. “DEP-5: Machine-readable debian/copyright”. In: *Machine-readable debian/copyright* (Feb. 24, 2012). URL: https://dep-team.pages.debian.net/deps/dep5/.

[18] *TeamsFTPMaster*. Mar. 13, 2020. URL: https://wiki.debian.org/TeamsFTPMaster.

[19] *CopyrightReviewTools*. Dec. 17, 2021. URL: https://wiki.debian.org/CopyrightReviewTolls.

[20] Mathew Palmer. “Debian-Mentors FAQ”. English. In: *Debian-Wiki* (2007). URL: https://wiki.debian.org/DebianMentorsFaq. GNU General Public License version 2.

[21] Jilayne et al. Lovejoy. “Open Source License Compliance Handbook”. In: *Open Source License* (Apr. 29, 2019). URL: https://github.com/finos/OSLC-handbook/tree/master/output.

[22] *Using Quilt*. July 22, 2020. URL: https://wiki.debian.org/UsingQuilt.

[23] Andreas Grünbacher. “How to Survive With Many Patches or Introduction to Quilt”. In: *Introduction to Quilt* (Feb. 22, 2012). URL: http://users.suse.com/~agruen/quilt.pdf.

[24] Raphael Hertzog. “DEP-3: Patch Tagging Guidelines”. In: *Patch Tagging Guidelines* (Nov. 26, 2009). URL: https://dep-team.pages.debian.net/deps/dep3/.

[25] *Git-Mailinfo -Manpage*. Apr. 20, 2020. URL: https://manpages.debian.org/buster/git-man/git-mailinfo.1.en.html.

[26] Niels Thykier et al. “Debian Policy for Java”. In: *Debian Java Policy* (July 27, 2020). URL: https://www.debian.org/doc/packaging-manuals/java-policy/.

[27] Torsten Werner twerner@debian.org Niels Thykier niels@thykier.net Javier Fernández-Sanguino Peña jfs@debian.org Sylvestre Ledru sylvestre@debian.org. "Debian Java FAQ." In: *Debian Java FAQ*. (May 22, 2014). URL: https://www.debian.org/doc/manuals/debian-java-faq/.

[28] *Help the Java Team distribute your project*. Jan. 31, 2019. URL: https://java.debian.net/blog/2019/01/help-the-java-team-distribute-your-project.html.

[29] Markus Koschany. "Packaging Java with Javatools". In: *https://people.debian.org/~apo/java/tutorial* (Aug. 2, 2018). URL: https://people.debian.org/%5Ctextasciitilde%7B%7Dapo/java/tutorial.html.

[30] *Introduction to the Standard Directory Layout*. Dec. 29, 2020. URL: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html.

[31] *Devsripts*. Nov. 28, 2020. URL: https://manpages.debian.org/unstable/devscripts/devscripts.1.en.html.

[32] *debian mit debootstrap in chroot-Umgebung installieren*. Feb. 16, 2014. URL: http://www.kai-hildebrandt.de/linux/debian%5C_chroot.html.

[33] wiki.debian.org. "Debian Releases". In: *https://wiki.debian.org/DebianReleases*. Nov. 29, 2020. URL: https://salsa.debian.org/debian/package-cycle/raw/master/package-cycle.svg.

[34] *SBuild*. Nov. 20, 2022. URL: https://wiki.debian.org/sbuild.

[35] *ReleaseTeam*. Mar. 22, 2021. URL: https://wiki.debian.org/Teams/releaseTeams.

Stichwortverzeichnis