

Ubuntu Paketerstellung und Veröffentlichung

Andreas Nicolai andreas.nicolai@gmx.net

Version 1.0 (10.08.2023)

Inhaltsverzeichnis

1. Überblick	1
2. Vorbetrachtungen	1
2.1. Upstream-Quelltext	2
2.2. Grundlegendes zu Paketen	3
2.3. Optionen für die Veröffentlichung von Paketen	4
3. Vorbereitung des Quelltextes für die Veröffentlichung als Paket	5
3.1. Erstellung von zusätzlichen Linux/Unix-spezifischen Dateien	6
3.1.1. Man-Pages	6
3.1.2. Application-Shortcut	7
3.1.3. Dateityp-Assoziation	8
3.2. Erweitern der CMake-Skripte mit <code>install()</code>	9
3.2.1. Testen der CMake-basierten Installation	12
4. Veröffentlichung als Open-Source Quellpaket	13
4.1. Vorbereitung	13
4.1.1. Launchpad Account	13
4.1.2. Signaturschlüssel (GPG) und SSH-Schlüssel erstellen und hochladen	13
4.1.3. Umgebungsvariablen einrichten	13
4.1.4. Launchpad-Account vervollständigen	15
4.1.5. Umgebungsvariablen	15
4.2. Arbeitsabläufe - Übersicht	15
4.3. Benötigte Pakete	15
4.3.1. Verzeichnisse und Bezeichnungen	16
4.4. Manuelle Erstellung eines neuen Debian-Source-Packages	16
4.4.1. Source-Quelltextarchiv erstellen	16
4.4.2. Vorbereiten des Debian-Pakets	19
4.4.3. Datei <code>debian/changelog</code>	20
4.4.4. Versionsnummernvergabe	21
4.4.5. Datei <code>debian/control</code>	21
4.4.6. Datei <code>debian/copyrights</code>	22
4.4.7. Datei <code>debian/rules</code>	23
4.5. Quell-Paket bauen	23
4.6. Quell-Paket prüfen	25
4.7. Binärpaket erstellen und prüfen	25
4.8. Veröffentlichung auf dem Launchpad PPA	27
5. Aktualisieren von Open-Source Quellpaketen	27
5.1. Neues Upstream-Release	27
5.2. Paketaktualisierung	28
6. Veröffentlichung als Binärpaket auf eigenem Server	28
6.1. Paketerstellung mit CPack	28
6.2. Hosting der Pakete in eigenem Repository	29

1. Überblick

In dieser Anleitung habe ich meine Erkenntnisse und Verfahrensweisen zur Erstellung von deb-Paketen für Ubuntu-Distributionen zusammengetragen. Am Beispiel von *MasterSim* beschreibe ich die verschiedenen Arbeitsschritte.

2. Vorbetrachtungen

Ubuntu ist eine klassische Linux-Distribution, welche in der jeweiligen Release-Version jeweils durch eine Auswahl von Software in bestimmten Versionen gekennzeichnet ist. Der elementare Vorteil von Distributionen ist, dass gemeinsame Bibliotheken von mehreren Paketen benutzt werden und so stets nur einmal auf der Platte liegen, und nicht wie bei Windows/macOS als Kopie mehrfach installiert werden. Dadurch ist das *MasterSim* Paket zum Schluss gerade mal 900 kB groß, da ja die C++ und Qt Laufzeitbibliotheken nicht nochmal erneut mitgeliefert werden müssen.

Weiterhin bietet dieses System den Vorteil, dass Fehlerbehebungen in gemeinsamen Bibliotheken sofort allen verwendenden Programmen zugute kommen, und diese nicht erst manuell aktualisiert und verteilt werden müssen. Daher ist es grundsätzlich eine gute Idee, existierende Pakete und Bibliotheken direkt zu nutzen, als diese im eigenen Projekt als Kopie zu duplizieren. Das geht nicht immer, aber dazu später mehr.

Dieser Vorteil von den gemeinsam genutzten Bibliotheken, oder allgemeiner noch *Paketen*, bedingt allerdings "etwas" mehr Arbeit. Sofern denn die zu verpackende Software Abhängigkeiten von installierten Softwarepaketen hat, muss man bei Änderungen der API solcher Bibliotheken entsprechend distributionsabhängige Quelltextmodifikationen berücksichtigen und auch für verschiedene Distro-Releases etwas veränderte Pakete erstellen.

Zusammenfassend die Vor- und Nachteile der Verwendung von Distributionen:

Vorteile:

- wesentlich geringerer Speicherplatzbedarf verglichen mit Softwarebundles (AppImage, Docker-Container, MacOS-AppBundles, Windows-Install-Paketen)
- einfache und schnelle Installation von Software und deren Aktualisierungen
- Verbesserungen/Bug-Fixes in verwendeten gemeinsamen Bibliotheken/Paketen fließen direkt ohne neue Release-Erstellung in abhängige Pakete ein

Nachteile:

- Aktualisierung von verwendeten Bibliotheken, weil z.B. ein neues Feature benötigt wird, ist in einer veröffentlichten Distribution in der Regel nicht möglich. Daher muss unter Umständen die Software manuell auf Verwendung einer neueren Bibliotheks-/Paketversion angepasst werden. Oder man verzichtet auf das Feature. Jedenfalls muss man bereits bei der Entwicklung ein Auge auf gewisse Abhängigkeiten haben und ggfs. im Quelltext Fallunterscheidungen



implementieren (= mehr Entwicklungsaufwand)

- Paketveröffentlichung muss für verschiedene Distributionen angepasst werden (= mehr Veröffentlichungsaufwand)

Schwierig wird es stets dann, wenn man sich entschließt, Bugfixes in der aktuellen Entwicklungsversion für eine ältere Distribution zurück zu portieren. Problematisch ist dies stets dann, wenn der Bugfix eine Funktion einer abhängigen Bibliothek verwendet, die in der Bibliotheksversion der älteren Distribution noch nicht verfügbar ist.

Damit ist man als Entwickler in einem gewissen Zwiespalt:

- einerseits sollte man versuchen, möglichst viele Distributionspakete nachzunutzen
- andererseits bringen lokale (und beliebig modifizierte) Kopien von externen Bibliotheken mehr Freiheiten für Entwickler und beschleunigen die Softwareentwicklung

Beispiel 1. Verwendung externer Bibliotheken bei MasterSim

Bei *MasterSim* wird z.B. die systemweit installierte *zlib* verwendet, jedoch *tinyxml* als eingebetteter, gepatchter und erweiterter Quelltext genutzt. Dies liegt zum einen an unseren IBK-spezifischen Erweiterungen (die man notfalls auch getrennt von der *tinyxml* lib ablegen könnte), und einigen Korrekturen/Verbesserungen im Quelltext von *tinyxml* selbst. Da die *tinyxml* lib selbst nicht mehr weiterentwickelt/gepflegt wird, können wir unsere Anpassungen/Verbesserungen nicht in den offiziellen Quelltext bekommen und müssen daher zwangsläufig eine veränderte, lokale Kopie halten (und diese manuell mit dem Original abgleichen).

Es ist also immer eine Grauzone und oft hängt die Entscheidung auch davon ab, inwieweit der Autor/Maintainer der Bibliothek offen für Erweiterungen/Änderungen ist.

2.1. Upstream-Quelltext

Der ursprüngliche Quelltext, der dem Paket zugrunde liegt, wird als **Upstream** bezeichnet.

Normalerweise wird, gerade bei plattformübergreifender Entwicklung, die Entwicklung in einem ausgewählten Distributionsrelease durchgeführt. Bspw. wird *MasterSim* gerade unter Ubuntu 20.04.3 LTS entwickelt. Entsprechend ist der Quelltext und das für die Release-Erstellung verwendete CMake-Buildsystem an diese Distribution angepasst.

MasterSim hat kaum Abhängigkeiten von installierten Bibliotheken oder Programmen, lediglich Qt5, cmake und zlib, und daher kann der Quelltext unverändert sowohl unter 18.04.6 LTS wie auch unter der aktuellsten 20.04.3 LTS kompiliert werden. Da sich die API der Bibliotheken nicht geändert hat, kann man sogar *MasterSim* nur für 18.04 kompilieren und die Binärdateien/Executables direkt unter 20.04 laufen lassen. Das erleichtert die Paketerstellung schon deutlich.



Allerdings hat CMake 3.16 bei Ubuntu 20.04 einen Automatismus beim `install()`-Befehl für die Zielverzeichniswahl, der bei CMake 3.10 in Ubuntu 18.04 fehlt. Daher

musste für die Erstellung unter 18.04 das `CMakeLists.txt`-Skript leicht angepasst werden. Dies ist nur ein Beispiel für gelegentlich notwendige Anpassungen im Quelltext und/oder Buildsystem.

Zum Überblick die Unterschiede in den Distro-Paket-Versionen:

Tabelle 1. Bibliotheksversionen in verschiedenen Distro-Releases

Bibliothek	Ubuntu 18.04 LTS	Ubuntu 20.04 LTS
cmake	3.10.2	3.16.3
Qt5	5.9.5	5.12.8

Die Änderungen in der Qt-Bibliothek haben keine Auswirkungen auf den C++-Quelltext. Sollten Funktionen verwendet werden, welche nur in neuen Qt-Versionen verfügbar sind, sollte man im Quelltext mit `#ifdef` entsprechende Fallunterscheidungen programmieren.

2.2. Grundlegendes zu Paketen

Ubuntu-Pakete sind `deb`-Dateien, welche eigentlich Debian-Pakete sind. Ubuntu baut auf Debian auf. Ein Debian-Paket für eine originale Debian-Distribution zu erstellen, ist wegen der stringenten Prüfanforderungen extrem aufwändig. Für Ubuntu ist es deutlich einfacher und da unsere Programme für Desktopanwender und weniger für Server gedacht sind, passen die auch besser in Ubuntu/Kubuntu oder ähnliche Distro rein.

Es gibt Binär- und Quellpakete. Binärpakete enthalten effektiv eine Kopie aller zu installierenden Dateien in der zukünftigen Verzeichnisstruktur. Bei Quellpaketen sind effektiv nur die Metadaten für die Erstellung und Verteilung des Pakets enthalten und ein Verweis auf ein beiliegendes Quelltextarchiv.

Man kann sich den Inhalt eines Binärpakets entweder grafisch mit dem Tool `gdebi-gtk` oder mittels `lesspipe` anzeigen lassen:

```
$ lesspipe mastersim_0.9.3-1_bionic_ppa1_amd64.deb
mastersim_0.9.3-1_bionic_ppa1_amd64.deb:
neues Debian-Paket, Version 2.0.
Größe 891372 Byte: control-Archiv= 1296 Byte.
    815 Byte,   15 Zeilen   control
    1944 Byte,  23 Zeilen   md5sums
Package: mastersim
Version: 0.9.3-1~bionic~ppa1
Architecture: amd64
Maintainer: Andreas Nicolai <andreas.nicolai@gmx.net>
Installed-Size: 3023
Depends: libc6 (>= 2.14), libgcc1 (>= 1:3.0), libqt5core5a (>= 5.9.0~beta), libqt5gui5 (>= 5.8.0), libqt5network5 (>= 5.0.2), libqt5printsupport5 (>= 5.0.2), libqt5widgets5 (>= 5.2.0), libstdc++6 (>= 5.2), zlib1g (>= 1:1.1.4)
Section: science
Priority: optional
Homepage: https://bauklimatik-dresden.de/mastersim
Description: FMI Co-Simulation Masterprogramm
 MasterSim is an FMI Co-Simulation master and programming library. It
 supports the Functional Mock-Up Interface for Co-Simulation in Version
 1.0 and 2.0. Using the functionality of version 2.0, it implements
```

various iteration algorithms that rollback FMU slaves and increase stability of coupled simulations.

*** Contents:

```
drwxr-xr-x root/root      0 2022-01-03 22:04 ./
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/bin/
-rwxr-xr-x root/root 1063592 2022-01-03 22:04 ./usr/bin/MasterSimulator
-rwxr-xr-x root/root 1727000 2022-01-03 22:04 ./usr/bin/MasterSimulatorUI
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/applications/
-rw-r--r-- root/root    272 2022-01-03 22:04 ./usr/share/applications/mastersim.desktop
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/doc/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/doc/mastersim/
-rw-r--r-- root/root    239 2022-01-03 22:04 ./usr/share/doc/mastersim/changelog.Debian.gz
-rw-r--r-- root/root   2676 2022-01-03 22:04 ./usr/share/doc/mastersim/copyright
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/icons/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/icons/hicolor/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/icons/hicolor/128x128/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/icons/hicolor/128x128/apps/
...
-rw-r--r-- root/root   4654 2022-01-03 18:03 ./usr/share/icons/hicolor/64x64/mimetypes/application-mastersim.png
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/locale/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/locale/de/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/locale/de/LC_MESSAGES/
-rw-r--r-- root/root  45907 2022-01-03 18:03 ./usr/share/locale/de/LC_MESSAGES/MasterSimulatorUI_de.qm
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/man/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/man/man1/
-rw-r--r-- root/root    689 2022-01-03 22:04 ./usr/share/man/man1/MasterSimulator.1.gz
-rw-r--r-- root/root    515 2022-01-03 22:04 ./usr/share/man/man1/MasterSimulatorUI.1.gz
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/mime/
drwxr-xr-x root/root      0 2022-01-03 22:04 ./usr/share/mime/packages/
-rw-r--r-- root/root    253 2022-01-03 22:04 ./usr/share/mime/packages/mastersim.xml
```

Die Dateiliste zeigt die zu installierenden Dateien und deren Zielpfade. Wie man an den Pfaden sieht, müssen ausführbare Dateien, Icons, aber auch Übersetzungsdateien oder man-pages in die jeweiligen Verzeichnisse kopiert werden. Wenn die Software dann als Paket-Version ausgeführt wird, muss entsprechend auf andere Pfade zugegriffen werden, als bei einer stand-alone Softwarearchiv-Installation. Dies erfordert eine Vorbereitung der Software für die Veröffentlichung.

2.3. Optionen für die Veröffentlichung von Paketen

Grundsätzlich kann man Debian-Pakete als quelloffene Pakete veröffentlichen und sie so für die Integration in das Ubuntu-Archiv vorbereiten. Außerdem kann man Launchpad nutzen, um automatisiert Pakete zu erstellen, zu aktualisieren und ein Privates Package Archive (PPA) zu hosten. Dieses Verfahren ist in [Abschnitt 4](#) beschrieben.

Alternativ kann man auch nur proprietäre Binar-Pakete erstellen und diese dann auf einem selbst gehosteten Paketrepository zur Verfügung stellen. Dieses Verfahren ist in [Abschnitt 6](#) beschrieben.

3. Vorbereitung des Quelltextes für die Veröffentlichung als Paket

MasterSim wird als Linux/MacOS/Windows Programm herausgegeben. Dabei werden folgende Dateitypen installiert, auf die innerhalb des Programms zugegriffen werden muss:

- Übersetzungsdateien (*.qm) für Qt und für das Programm selbst
- Beispiele und Datenbankdateien (letzteres braucht *MasterSim* noch nicht)

Andere Dateien werden nur für die Systemintegration benötigt und müssen vom jeweiligen Installer (Inno-Setup unter Windows, dmg für Mac, deb-Paket unter Linux oder manuell bei 7z-Archiven) im System verankert werden.

Damit der Quelltext weitgehend ahnungslos hinsichtlich der Plattform oder Installationsmethode bleiben kann, werden alle Pfade durch eine einheitliche Verzeichnis-Klasse (`MSIMDirectories` oder allgemein bei Programmen mit QtExt-Bibliotheksunterstützung `QtExt::Directories`) bereitgestellt.

Die Funktion `resourcesRootDir()` liefert das Basisverzeichnis für die im Programm verwendeten Ressourcen. Da die Verzeichnisse für die Übersetzungsdateien selbst von der gewählten Sprache abhängen, haben diese Funktionen die Syntax `translationsFilePath(langID)` bzw. `qtTranslationsFilePath(langID)`.

Tabelle 2. Verzeichnispfade relativ zur ausführbaren Datei für verschiedene Ressourcen and Installationsvarianten

Plattform*	Pfad/Funktion	relativer Pfad
Linux - deb		
	<code>resourcesRootDir()</code>	<code>../share/mastersim</code>
	<code>translationsFilePath(langID)</code>	<code>../share/locale/<langID>/LC_MESSAGES/MasterSimulatorUI.qm</code>
	<code>qtTranslationsFilePath(langID)</code>	<code>/usr/share/qt5/translations/qt_<langID>.qm</code>
Linux - 7z		
	<code>resourcesRootDir()</code>	<code>../resources</code>
	<code>translationsFilePath(langID)</code>	<code>../resources/translations/MasterSimulatorUI_<langID>.qm</code>
	<code>qtTranslationsFilePath(langID)</code>	<code>/usr/share/qt5/translations/qt_<langID>.qm</code>
Windows		
	<code>resourcesRootDir()</code>	<code>../resources</code>
	<code>translationsFilePath(langID)</code>	<code>../resources/translations/MasterSimulatorUI_<langID>.qm</code>
	<code>qtTranslationsFilePath(langID)</code>	<code>../resources/translations/qt_<langID>.qm</code>



Unter Windows werden die Qt5 Bibliotheken und passend dazu die `qt_de.qm` im

Installer mitgeliefert. Daher liegt hier die Qt-Übersetzungsdatei immer unterhalb `resources`. Unter Linux muss die `qt_de.qm` mit der jeweils installierten Qt-Version übereinstimmen, weswegen unabhängig von der Installationsvariante (deb oder stand-alone 7z) *immer* die systemweit installierte Übersetzungsdatei verwendet wird.

Das `resourcesRootDir()` setzt sich bei der deb-Paket-Variante aus dem Präfix `../share` und dem Paketnamen zusammen, hier `mastersim`.

Eine weitere Besonderheit besteht bei der deb-Paket-Installation darin, dass die Übersetzungsdateien für das Programm *kein* Suffix `_de` haben. Darauf muss man beim Formulieren der `install()` Regeln im CMake achten.

Die Entscheidung darüber, welche Pfade für Ressourcen und Übersetzungsdateien verwendet werden, wird zur Compile-Zeit getroffen. Relevant dafür sind die Defines `Q_OS_LINUX` zur Auswahl des qt-Übersetzungsdatei-Pfads und `IBK_BUILDING_DEBIAN_PACKAGE` zur Konfiguration der Suchpfade entsprechend systemweiter Installation. Letzteres sollte zu Testzwecken via Kommandozeile dem cmake-Programm übergeben werden, siehe auch [Abschnitt 3.2.1](#).

3.1. Erstellung von zusätzlichen Linux/Unix-spezifischen Dateien

3.1.1. Man-Pages

Wenn man ausführbare Dateien ausliefert, sollte man dazu passende man-Seiten ausliefern. Diese sind Text-Dateien mit einfachen Formatangaben (aus Zeiten lange vor Markdown oder ähnlichem).

Idealerweise spucken unsere `IBK::Argparser`-basierten Programme mittels `--man-page` solche Seiten automatisch aus, aber leider ist diese Funktionalität nie ausprogrammiert worden. Sonst würde ein:

```
$ ./MasterSimulator --man-page > MasterSimulator.1
```

bereits eine gültige und aktuelle Man-page erzeugen.

Stattdessen macht man das vorläufig noch manuell, z.B. mit `help2man` oder `txt2man`. Für `help2man` müsste man die Ausgabe von `--help` wahrscheinlich noch etwas an den geforderten Standard anpassen. Deshalb habe ich `txt2man` verwendet und die entstandene Datei noch minimal nachbearbeitet:

```
$ ./MasterSimulator --help | txt2man > MasterSimulator.1
```



Diese bash-Kommandozeile führt zunächst `./MasterSimulator --help` aus, welches die Hilfeseite in die Ausgabe schiebt. Das `|` Zeichen führt dazu, dass diese Ausgabe nun als Eingabestream dem Tool `txt2man` zur Verfügung gestellt wird, welches seinerseits die daraus generierte man-Seite in die Ausgabe schreibt. Zum Schluss leitet `> MasterSimulator.1` die Ausgabe noch in die Datei um.

Die Dateierweiterung ".1" deutet auf die Sektion des Programmes und man-page hin - 1 steht hier für reguläre Programme/Tools.

Die generierte Man-Seite muss man noch bearbeiten, zumindest die Kopfzeile:

```
.TH "MASTERSIMULATOR" "1" "January 01, 2022" "0.9.1" "mastersim"
```

und gegebenenfalls noch einige Stellen im Text.



Hier sieht man auch schon ein Problem: bei jedem Release müsste diese Kopfzeile um die aktuelle Versionsnummer aktualisiert werden, und auch das Datum sollte jeweils erneuert werden. Dies alles spricht dafür, dass man die Option `--man-page` fertig implementiert und dann als post-build-Schritt vor der Installation ausführt (TODO Stefan oder Andreas!).

Die Man-pages für *MasterSimulator* und *MasterSimulatorUI* gehören in die jeweiligen Unterverzeichnisse, also:

```
MasterSimulator/doc/MasterSimulator.1  
MasterSimulatorUI/doc/MasterSimulatorUI.1
```

3.1.2. Application-Shortcut

Um im Programmstarter *MasterSim* angezeigt zu bekommen (und danach suchen zu können), muss man eine `.desktop` Datei erstellen.

Dieser sieht für MasterSimulatorUI so aus:

`mastersim.desktop`

```
[Desktop Entry]  
Name=MasterSim  
GenericName=FMI Co-Simulation Master  
Comment=FMI Co-Simulations Master  
Keywords=FMI;FMU;Simulation  
Exec=MasterSimulatorUI %f  
Icon=mastersim  
Terminal=false  
Type=Application  
Categories=Science  
StartupNotify=true  
MimeType=application/mastersim
```

Letztlich definiert diese Datei den Namen, ein paar Schlüsselworte, ob ein Terminalfenster gebraucht wird oder nicht, den Icon-Namen `mastersim` (wichtig, kein absoluter Pfad hier!), ein paar Kategorisierungsinfos und natürlich die auszuführende Datei im Schlüsselwort `Exec`.

Das **Argument %f** sagt dabei, dass ein über Dateityp-Assoziation verknüpfter Dateipfad hier übergeben

wird. D.h. wenn man im Dateimanager auf eine `.msim`-Datei doppelklickt (oder "Öffnen mit..."-auswählt), wird MasterSim mit dieser Datei als Argument gestartet. Wie man diese Verknüpfung definiert, wird gleich erklärt, wichtig ist hierbei aber die Definition des MIME-Typs als `application/mastersim`.

Wichtig beim Icon und Exec Eintrag: es werden keine absoluten Pfade definiert. Das Linux-System erwartet die Installation der ausführbaren Datei in einem Suchpfad des Systems. Das Icon wird in einem der Standard-Verzeichnisse für Icons gesucht, unter dem Namen `mastersim.<Bildtyp>` (dabei können verschiedene Dateierweiterungen verwendet werden, weswegen man auch auf die Dateierweiterung verzichtet).

Die Datei `mastersim.desktop` wird unter `MasterSimulatorUI/resources/mastersim.desktop` gespeichert.

3.1.3. Dateityp-Assoziation

Damit im System die `msim`-Dateien entsprechend mit einem Icon dekoriert werden und via Doppelklick die Anwendung geöffnet wird, muss man sogenannte MIME-Typen verknüpfen. Dazu erstellt man eine Datei `masterim.xml`:

`mastersim.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<mime-info xmlns='http://www.freedesktop.org/standards/shared-mime-info'>
  <mime-type type="application/mastersim">
    <comment>MasterSim project file</comment>
    <glob pattern="*.msim"/>
  </mime-type>
</mime-info>
```

In dieser Datei findet sich der in der `.desktop`-Datei definierte MIME-Typ `application/mastersim` wieder. Außerdem wird das Dateisuchmuster als `*.msim` festgelegt (es könnten hier auch weitere Dateierweiterungen assoziiert werden, indem man mehrere `<glob>` Elemente definiert. Außerdem können Dateitypen anhand eines "Magic-Headers" erkannt werden, wie er z.B. bei unseren `d6o/d6b` und Verwendung findet. Siehe dazu <https://specifications.freedesktop.org/shared-mime-info-spec> und die darin verlinkte Spezifikation).



In der XML-Datei muss aus Kompatibilitätsgründen die mime-info URL <http://www.freedesktop.org/standards/shared-mime-info> stehen, auch wenn die Spezifikation aktuell unter <https://specifications.freedesktop.org/shared-mime-info-spec> zu finden ist (sonst bekommt man eine Fehlermeldung beim Aktualisieren der MIME-Datenbank).

Die Datei `mastersim.xml` wird unter `MasterSimulatorUI/resources/mastersim.desktop` gespeichert.



Bei der Installation werden diese Dateien an die entsprechenden Orte im Dateisystem kopiert. Dies alleine reicht aber noch nicht aus, um Anwendung und Dateiverknüpfung im System bekannt zu machen. Dafür müssen noch Skripte gestartet werden, welche die jeweiligen Datenbanken aktualisieren. Netterweise macht das die

Paketverwaltung bei Installation eines deb-Pakets automatisch für uns.

3.2. Erweitern der CMake-Skripte mit `install()`

CMake bietet eine recht komfortable Möglichkeit, nach dem Erstellen (`make`) auch alle Dateien an die richtige Stelle zu installieren. So kann man mit:

```
cmake ..  
make  
sudo make install
```

die Anwendung auch direkt aus dem Quelltextarchiv ins System installieren. Allerdings kann man so kein sinnvolles "uninstall" machen, d.h. einmal installierte Dateien müsste man händisch wieder aus den verschiedenen Installationspfaden löschen. Debian-Pakete sind hier sinnvoller, da diese bei Aktualisierungen oder De-Installation automatisch vorher installierte und nicht mehr benötigte Dateien entfernt.

Da die Installation via `cmake/make` nur unter Linux sinnvoll ist (unter Windows gibt's sinnige Installer und unter MacOS kapseln die App-Bundles sowieso alles), sollten entsprechende `install()` Aufrufe im `CMakeLists.txt` Skript in `if`-Blöcken stehen:

Install-Abschnitt aus der Datei `MasterSimulator/projects/cmake_local/CMakeLists.txt`

```
if (UNIX AND NOT APPLE)  
  
    # installation targets for Unix systems  
    include(GNUInstallDirs)  
  
    # MasterSimulator -> /usr/bin  
    install(TARGETS ${PROJECT_NAME} RUNTIME DESTINATION bin )  
  
    # Man-page  
    install(FILES ${PROJECT_SOURCE_DIR}/../doc/${PROJECT_NAME}.1 DESTINATION ${CMAKE_INSTALL_MANDIR}/man1 )  
  
endif (UNIX AND NOT APPLE)
```

In diesem Skript wird der Platzhalter `${PROJECT_NAME}` durch `MasterSimulator` ersetzt.

Letztlich müssen zwei `CMakeLists.txt`-Dateien angepasst werden.

Die Erweiterung für `MasterSimulator/projects/cmake_local/CMakeLists.txt` ist oben bereits gezeigt. Lediglich die ausführbare Datei `MasterSimulator` wird ins `bin`-Verzeichnis installiert (welches je nach Installationspräfix `/usr/bin` oder `/usr/local/bin` ist).

`include(GNUInstallDirs)` definiert diverse Installationpräfixes, wie z.B. `${CMAKE_INSTALL_MANDIR}`. Bei CMake 3.10 muss man bei zu installieren build-targets (hier eine ausführbare Datei, könnte aber auch eine Bibliothek sein) noch explizit das Zielverzeichnis definieren (hier `bin`). Ab CMake 3.16 wird das automatisch je nach Typ des "Targets" erkannt. Da `MasterSim` aber auch unter Ubuntu 18.04 (mit CMake 3.10) funktionieren soll, steht der Zielpfad nochmal explizit da.

Für die Programmoberfläche **MasterSimulatorUI** wird etwas mehr benötigt:

*Install-Abschnitt aus der Datei **MasterSimulator/projects/cmake_local/CMakeLists.txt***

```
# Support for 'make install' on Unix/Linux (not on MacOS!)
if (UNIX AND NOT APPLE)

    # installation targets for Unix systems
    include(GNUInstallDirs)

    # MasterSimulator -> /usr/bin
    install(TARGETS ${PROJECT_NAME} RUNTIME DESTINATION bin )

    # Man-page
    install(FILES ${PROJECT_SOURCE_DIR}/../..doc/${PROJECT_NAME}.1
        DESTINATION ${CMAKE_INSTALL_MANDIR}/man1 )

    # Translation files
    install(FILES ${PROJECT_SOURCE_DIR}/../..resources/translations/${PROJECT_NAME}_de.qm
        DESTINATION ${CMAKE_INSTALL_LOCALEDIR}/de/LC_MESSAGES/
        RENAME ${PROJECT_NAME}.qm)

    # Desktop file
    install(FILES ${PROJECT_SOURCE_DIR}/../..resources/mastersim.desktop
        DESTINATION ${CMAKE_INSTALL_DATAROOTDIR}/applications )

    # Mime type
    install(FILES ${PROJECT_SOURCE_DIR}/../..resources/mastersim.xml
        DESTINATION ${CMAKE_INSTALL_DATAROOTDIR}/mime/packages )

    ....
```

Der erste Teil der Installation von **MasterSimulatorUI** ist identisch mit dem des Konsolensolvers **MasterSimulator**. Interessant wird es bei der Übersetzungsdatei.

Die Anwendungsübersetzungsdatei liegt in **MasterSimulatorUI/resources/translations/MasterSimulatorUI_de.qm** und muss nach **/usr/share/locale/de/LC_MESSAGES/MasterSimulatorUI.qm** kopiert werden. ACHTUNG: der Dateiname ändert sich! Der Pfad **/usr/share/locale/** wird wiederum als Platzhalter **\${CMAKE_INSTALL_LOCALEDIR}** zur Verfügung gestellt. Die Umbenennung macht man mit dem **RENAME** Befehl innerhalb der **install()** Funktion. Dieser muss immer als letztes angegeben werden.

Danach werden die **mastersim.desktop** und **mastersim.xml** Dateien in die jeweiligen Zielpfade installiert.

Weiter geht es mit den Anwendungsicons:

*Zweiter Teil des Install-Abschnitts aus der Datei **MasterSimulator/projects/cmake_local/CMakeLists.txt***

```
....

# Icons
set(ICON_ROOT_DIR ${CMAKE_INSTALL_DATAROOTDIR}/icons/hicolor)

# Anwendungsicons
install(FILES ${PROJECT_SOURCE_DIR}/../..resources/gfx/logo/Icon_512.png
    DESTINATION ${ICON_ROOT_DIR}/512x512/apps
```

```

    RENAME mastersim.png)
install(FILE ${PROJECT_SOURCE_DIR}/../../resources/gfx/logo/Icon_256.png
    DESTINATION ${ICON_ROOT_DIR}/256x256/apps
    RENAME mastersim.png)

...

install(FILE ${PROJECT_SOURCE_DIR}/../../resources/gfx/logo/Icon_16.png
    DESTINATION ${ICON_ROOT_DIR}/16x16/apps
    RENAME mastersim.png)

# Mime-type Icons
install(FILE ${PROJECT_SOURCE_DIR}/../../resources/gfx/logo/Icon_512.png
    DESTINATION ${ICON_ROOT_DIR}/512x512/mimetypes
    RENAME application-mastersim.png)

install(FILE ${PROJECT_SOURCE_DIR}/../../resources/gfx/logo/Icon_256.png
    DESTINATION ${ICON_ROOT_DIR}/256x256/mimetypes
    RENAME application-mastersim.png)

...

install(FILE ${PROJECT_SOURCE_DIR}/../../resources/gfx/logo/Icon_16.png
    DESTINATION ${ICON_ROOT_DIR}/16x16/mimetypes
    RENAME application-mastersim.png)

endif (UNIX AND NOT APPLE)

```

Auch hier werden die Icons wieder umbenannt, da sie in unterschiedlichen Verzeichnissen liegen, aber alle den gleichen Dateinamen haben.

Der oben gekürzte Textblock wäre ziemlich lang, wenn alle Icongrößen installiert würden. Die einzelnen `install()` Anweisungen sind bis auf die Icon-Größe identisch. CMake erlaubt es einem, solche Abschnitte durch Verwendung von Schleifen zu vereinfachen.

Verbesserter Install-Abschnitt für Icons aus der Datei

MasterSimulator/projects/cmake_local/CMakeLists.txt

```

....

# Icons
set(ICON_ROOT_DIR ${CMAKE_INSTALL_DATAROOTDIR}/icons/hicolor)

foreach(ICON_SIZE 512 256 64 48 32 16)
    install(FILE ${PROJECT_SOURCE_DIR}/../../resources/gfx/logo/Icon_${ICON_SIZE}.png
        DESTINATION ${ICON_ROOT_DIR}/${ICON_SIZE}x${ICON_SIZE}/apps
        RENAME mastersim.png)

    install(FILE ${PROJECT_SOURCE_DIR}/../../resources/gfx/logo/Icon_${ICON_SIZE}.png
        DESTINATION ${ICON_ROOT_DIR}/${ICON_SIZE}x${ICON_SIZE}/mimetypes
        RENAME application-mastersim.png)
endforeach()

endif (UNIX AND NOT APPLE)

```

3.2.1. Testen der CMake-basierten Installation

Wenn man jetzt des Quelltextarchiv mit `make` erstellt hat und nachfolgend `make install` aufruft, versucht CMake die Dateien standardmäßig nach `/usr/local` zu installieren. Dafür wären zum einen Superuser-Rechte notwendig, also `sudo make install`. Zum Testen sollte man sich aber nicht das System zumüllen, weswegen man eher ein Test-Install-Präfix wählen sollte:

```
$ mkdir bb-test
$ cd bb-test
$ cmake -DCMAKE_INSTALL_PREFIX=/home/ghorwin/tmp ..
$ make
...
```

Durch Definition des Arguments `CMAKE_INSTALL_PREFIX` legt man das Basisverzeichnis (analog zu `/usr` oder `/usr/local`) fest.

Vorher sollte man noch in der obersten `CMakeLists.txt`-Datei die Zeile

```
add_definitions( -DIBK_BUILDING_DEBIAN_PACKAGE )
```

einfügen, welches die Software im deb-Installmodus konfiguriert (Pfade für Übersetzungsdateien und Ressourcen/Beispiele).

Das Ausführen von `make install` führt zu folgender Ausgabe:

```
$ make install
[ 3%] Built target minizip
[ 17%] Built target IBK
[ 18%] Built target IBKMK
[ 32%] Built target BlockMod
[ 34%] Built target TiCPP
[ 41%] Built target MasterSim
[ 42%] Built target MasterSimulator
[ 93%] Built target MasterSimulatorUI
[ 94%] Built target Math003Part1
[ 96%] Built target Math003Part2
[ 97%] Built target Math003Part3
[ 98%] Built target LotkaVolterraPrey
[100%] Built target LotkaVolterraPredator
Install the project...
-- Install configuration: "RelWithDebInfo"
-- Installing: /home/ghorwin/tmp/bin/MasterSimulator
-- Installing: /home/ghorwin/tmp/share/man/man1/MasterSimulator.1
-- Installing: /home/ghorwin/tmp/bin/MasterSimulatorUI
-- Installing: /home/ghorwin/tmp/share/man/man1/MasterSimulatorUI.1
-- Installing: /home/ghorwin/tmp/share/locale/de/LC_MESSAGES/MasterSimulatorUI.qm
-- Installing: /home/ghorwin/tmp/share/applications/mastersim.desktop
-- Installing: /home/ghorwin/tmp/share/mime/packages/mastersim.xml
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/512x512/apps/mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/512x512/mimetypes/application-mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/256x256/apps/mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/256x256/mimetypes/application-mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/64x64/apps/mastersim.png
```

```
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/64x64/mimetypes/application-mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/48x48/apps/mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/48x48/mimetypes/application-mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/32x32/apps/mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/32x32/mimetypes/application-mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/16x16/apps/mastersim.png
-- Installing: /home/ghorwin/tmp/share/icons/hicolor/16x16/mimetypes/application-mastersim.png
```

Scheint also alles zu stimmen. Nun kann man das Programm ausführen und prüfen, ob die Übersetzungsdateien und sonstige Ressourcen korrekt gefunden werden:

```
$ /home/ghorwin/tmp/bin/MasterSimulatorUI
App translation file path = '/home/ghorwin/tmp/bin/./share/locale/de/LC_MESSAGES/MasterSimulatorUI.qm'.
Qt translation file path = '/usr/share/qt5/translations/qt_de.qm'.
Qt translation file loaded successfully
Application translator loaded successfully
```

Passt. Nun ist alles fertig für die Erstellung von Debian-Paketen.

4. Veröffentlichung als Open-Source Quellpaket

4.1. Vorbereitung

4.1.1. Launchpad Account

- Account erstellen: <https://launchpad.net>

4.1.2. Signaturschlüssel (GPG) und SSH-Schlüssel erstellen und hochladen

- Anleitung für GPG und SSH folgen: <https://packaging.ubuntu.com/html/getting-set-up.html>

4.1.3. Umgebungsvariablen einrichten

```
export DEBFULLNAME="Bob Dobbs"
export DEBEMAIL="subgenius@example.com"
```

GPG Schlüssel

Neuen Schlüssel erstellen:

```
$ gpg --gen-key

gpg: Schlüssel 6E0814BD3FCA8338 ist als ultimativ vertrauenswürdig gekennzeichnet
gpg: Verzeichnis '/home/ghorwin/.gnupg/openpgp-revocs.d' erzeugt
gpg: Widerrufszertifikat wurde als '/home/ghorwin/.gnupg/openpgp-revocs.d/34FC6FB934502913B4C1DCA86E0814BD3FCA8338.rev'
gespeichert.
Öffentlichen und geheimen Schlüssel erzeugt und signiert.
```

```
pub  rsa3072 2021-12-29 [SC] [verfällt: 2023-12-29]
     34FC6FB934502913B4C1DCA86E0814BD3FCA8338
uid      Andreas Nicolai <andreas.nicolai@gmx.net>
sub  rsa3072 2021-12-29 [E] [verfällt: 2023-12-29]
```

Die Schlüssel-ID ist **6E0814BD3FCA8338**. Schlüssel auf Schlüsselserver hochladen:

```
$ gpg --send-keys --keyserver keyserver.ubuntu.com 6E0814BD3FCA8338
gpg: sende Schlüssel 6E0814BD3FCA8338 auf hkp://keyserver.ubuntu.com
```

Eigenen Fingerabdruck finden, der mit der eigenen E-Mailadresse verknüpft ist:

```
$ gpg --fingerprint andreas.nicolai@gmx.net
gpg: "Trust-DB" wird überprüft
gpg: marginals needed: 3 completes needed: 1 trust model: pgp
gpg: Tiefe: 0 gültig: 2 signiert: 0 Vertrauen: 0-, 0q, 0n, 0m, 0f, 2u
gpg: nächste "Trust-DB"-Pflichtüberprüfung am 2023-12-29
pub  rsa3072 2021-12-29 [SC] [verfällt: 2023-12-29]
     34FC 6FB9 3450 2913 B4C1 DCA8 6E08 14BD 3FCA 8338
uid      [uneingeschränkt] Andreas Nicolai <andreas.nicolai@gmx.net>
sub  rsa3072 2021-12-29 [E] [verfällt: 2023-12-29]
```

Prüfen, ob der Schlüssel auf dem Schlüsselserver hochgeladen ist (Verteilung auf Schlüsselserver dauert manchmal etwas):

```
$ gpg --keyserver hkp://keyserver.ubuntu.com --search-key 'andreas.nicolai@gmx.net'
gpg: data source: http://162.213.33.9:11371
(1) Andreas Nicolai <andreas.nicolai@gmx.net>
     3072 bit RSA key 6E0814BD3FCA8338, erzeugt: 2021-12-29
Keys 1-1 of 1 for "andreas.nicolai@gmx.net".
```

GPG-Key zu Thunderbird hinzufügen, damit man Bestätigungsmail entschlüsseln kann

GPG-Schlüssel exportieren:

```
> gpg --export-secret-keys --armor > my-secret-keys.asc
```

- → In Thunderbird diesen Schlüssel importieren
- → Account-Einstellungen
- → Ende-zu-Ende-Verschlüsselung
- → OpenPGP → Schlüssel hinzufügen
- → Schlüssel auswählen, fertig

ssh-Schlüsselerstellung

ssh-key (RSA, 4096 Bits) neu erstellen:

```
$ ssh-keygen -t rsa -b 4096
```

4.1.4. Launchpad-Account vervollständigen

- GPG Schlüssel hinzufügen (Bestätigungsmail lesen/entschlüsseln)
- SSH-Schlüssel hinzufügen
- Account Bild/Metadaten
- eigenes PPA erstellen, Beispielsweise "sim"

Web-URL: <https://launchpad.net/~ghorwin/+archive/ubuntu/sim>

```
# Hinzufügen des Repos auf jedem beliebigen Rechner
# (geht sobald Pakete veröffentlicht sind und der Signaturschlüssel
# auf die Keyserver verteilt wurde)
$ sudo add-apt-repository ppa:ghorwin/sim

# beim nächsten "sudo apt update" wird auch im ppa nach Paketen für
# aktuelle Distro gesucht
```

4.1.5. Umgebungsvariablen

in `.profile` oder `.bashrc`:

```
export DEBFULLNAME="Andreas Nicolai"
export DEBEMAIL="andreas.nicolai@gmx.net"
```

4.2. Arbeitsabläufe - Übersicht

1. komplett neues Quellpaket erstellen
2. Paket aktualisieren (ohne Quelltextänderung; nur Paketfehler beheben/Paket verbessern)
3. Paket aktualisieren wegen Upstream-Release-Update (neuer Upstream-Quelltext)
4. fertiges Paket für eine andere Distributionsversion veröffentlichen

4.3. Benötigte Pakete

```
$ sudo apt install debhelper lintian
```

4.3.1. Verzeichnisse und Bezeichnungen

Für die nachfolgenden Schritte werden verschiedene Verzeichnisse benötigt.

- *Basisverzeichnis* : das Verzeichnis (innerhalb eines Versionskontrollsystems), welches für jede unterstützte Distribution ein Unterverzeichnis enthält.
- *Distro-Verzeichnis* : das Verzeichnis (innerhalb eines Versionskontrollsystems), welches alle für die Erstellung/Aktualisierung eines Pakets für *eine bestimmte Distributionsversion* benötigt; in diesem Verzeichnis wird das Arbeitsverzeichnis erstellt
- *Arbeitsverzeichnis* : ein temporäres Verzeichnis, in dem alle Dateien für die Erstellung des Releases hineinkopiert/generiert werden; dieses Verzeichnis enthält zum Schluss die erstellten Pakete

Beispiel 2. Verzeichnisstruktur (enthält einige temporäre Dateien und Verzeichnisse)

```

debPackaging
├── CMakeLists.txt
├── MasterSim-git
├── mastersim-0.9.2
├── mastersim_0.9.2.orig.tar.xz
├── scripts
│   ├── extractVersion.py
│   ├── update_repo.sh
│   └── update_source_code_archive.sh
├── ubuntu-18.04-bionic
│   └── ...
├── ubuntu-20.04-focal
│   └── *Distro-Verzeichnis*, enthält Dateien für Ubuntu 20.04 Focal
│       ├── debian
│       │   ├── changelog
│       │   ├── control
│       │   ├── copyright
│       │   ├── rules
│       │   └── source
│       │       └── format
│       ├── mastersim-0.9.2
│       │   └── Distro-spezifisches *Arbeitsverzeichnis*
│       ├── mastersim_0.9.2-1~focal~ppa1.debian.tar.xz
│       ├── mastersim_0.9.2-1~focal~ppa1.dsc
│       ├── mastersim_0.9.2-1~focal~ppa1_source.buildinfo
│       ├── mastersim_0.9.2-1~focal~ppa1_source.changes
│       ├── mastersim_0.9.2.orig.tar.xz
│       │   └── symlink auf top-level Archiv
│       └── readme.md
├── ubuntu-21.10-impish
│   └── ...

```

Nachfolgend sind alle einzelnen Schritte zu Erstellung eines Source-Pakets beschrieben. Dabei werden alle in dieser Verzeichnishierarchie gelisteten Dateien und Verzeichnisse besprochen.

4.4. Manuelle Erstellung eines neuen Debian-Source-Packages

4.4.1. Source-Quelltextarchiv erstellen

Erster Schritt ist die Erstellung eines Quelltextarchives. Dies erfolgt im Basisverzeichnis, da der

Upstream-Quelltext für alle zu erstellenden Pakete gleich ist.

```
# Im Basisverzeichnis auszuführen
git clone https://github.com/ghorwin/MasterSim.git MasterSim-git
```

Es entsteht das Verzeichnis **MasterSim-git** parallel zur den Distro-Verzeichnissen.

Falls das Verzeichnis schon existiert, reicht auch ein

```
git pull --rebase
```

im **MasterSim-git**-Verzeichnis.

Wenn man den Zeitaufwand für das ständig neu clonen vermeiden will, hilft folgendes Script:

```
echo "*** STEP 1 : cloning MasterSim.git ***" &&
if [ ! -d "MasterSim-git" ]; then
  echo "Cloning github repo"
  git clone https://github.com/ghorwin/MasterSim.git MasterSim-git
else
  echo "Reverting local changes and pulling newest revisions from github"
  (cd MasterSim-git && git reset --hard HEAD && git clean -fdx && git pull --rebase)
fi &&
du -h --summarize MasterSim-git/
```

Kopie des Repo-Quelltextes erstellen

Das Name des Paketverzeichnis ergibt sich aus der Upstream-Versionsnummer und dem Paketnamen. Die Upstream-Versionsnummer wird aus der Datei `MSIM_Constants.cpp` entnommen, aktuell 0.9.2.

Man erstellt neben dem git-Clone-Verzeichnis das Quelltextverzeichnis, beispielsweise **mastersim-0.9.2**.

Jetzt werden die Daten aus dem Quelltextverzeichnis in das Paketverzeichnis kopiert:

```
# Ausführen im Arbeitsverzeichnis
echo "*** STEP 2 : Copy source directory ***" &&
rsync -a --delete --exclude=".*" MasterSim-git/ mastersim-0.9.2 &&
du -h --summarize mastersim-0.9.2/
```

Bei diesem Befehl werden alle versteckten Dateien (also primär das Verzeichnis **.git** und alle **.gitignore**) weggelassen.

Nun ist das Verzeichnis **mastersim-0.9.2** ein reines Quelltextverzeichnis.



Das Quelltextverzeichnis liegt aktuell neben dem **MasterSim-git**-Verzeichnis. Mitunter ist es notwendig, Quelltextanpassungen (Patches) für bestimmte Distributionen einzuarbeiten, wenn sich der Upstream-Quelltext nicht problemlos kompilieren lässt.

Dazu werden dann Patch-Dateien angewendet. Mehr dazu hier <https://packaging.ubuntu.com/html/patches-to-packages.html>.

Quelltextverzeichnis bereinigen und Top-Level CMakeLists.txt einfügen

Dateien, welche für das Erstellen und/oder Nachvollziehen des Quelltextes nicht zwingend notwendig sind, können entfernt werden. Auch sollten Dateien, welche später Probleme bei den Paket-Sicherheitschecks bringen, entfernt werden. Allgemein sollte man versuchen, den Platzbedarf für das Quelltextarchiv so klein wie möglich zu halten.

```
echo "*** STEP 3 : Cleaning out source directory ***" &&
rm -rf mastersim-0.9.2/third-party &&
rm -rf mastersim-0.9.2/doc &&
mv $TARGETDIR/data/examples/linux64 $TARGETDIR/examples &&
rm -rf $TARGETDIR/data &&
mkdir --parents $TARGETDIR/data/examples/ &&
mv $TARGETDIR/examples $TARGETDIR/data/examples/linux64 &&
rm -rf mastersim-0.9.2/cross-check &&
rm -rf mastersim-0.9.2/externals/zlib &&
du -h --summarize mastersim-0.9.2/
```



Das `CMakeLists.txt` install script erwartet (später) die zu installierenden Beispiele im Verzeichnis `data/examples/linux64`. Statt nun alle anderen Verzeichnisse und Dateien individuell zu löschen, "retten" wir das Verzeichnis zuerst mit `mv` und schieben es zum Schluss wieder an die originale Position.

Es wird nun noch die Top-Level `CMakeLists.txt` benötigt, welche aber nur ein minimaler Wrapper um die eigentliche CMake-Datei `build/cmake/CMakeLists.txt`-Datei ist. Außerdem wird in dieser Wrapper-Datei noch das Define für Debian-Paket-Erstellung gesetzt:

```
project( MasterSimDebPackage )
cmake_minimum_required( VERSION 3.10 )

# Debian package build flag for IBK-based applications
if (UNIX AND NOT APPLE)
    ADD_DEFINITIONS( -DIBK_BUILDING_DEBIAN_PACKAGE )
endif (UNIX AND NOT APPLE)

add_subdirectory( build/cmake MasterSimPackage)
```



Man könnte auch im Upstream-Quelltext bereits eine Top-Level `CMakeLists.txt`-Datei haben. Dann müsste diese aber eine Option zum Einschalten des Defines enthalten.

Quelltext-Archiv packen

Das Quelltextverzeichnis wird nun gepackt und `mastersim-0.9.2.orig.tar.xz` benannt. Man könnte auch ein `tar.gz` oder `tar.bz2` erstellen, aber `xz` komprimiert ganz gut.

```
echo "*** STEP 4 : Creating source tarball ***" &&
tar cf - mastersim-0.9.2/ | xz -z - > mastersim_0.9.2.orig.tar.xz &&
du -h --summarize mastersim_0.9.2.orig.tar.xz
```

Dieses Quelltextarchiv ist nun die Grundlage für neue Pakete und für Aktualisierungen von Paketen ohne Quelltextänderung.



Das Format des Dateinamens `mastersim_0.9.2.orig.tar.xz` ist wichtig. Die Debian-Paketerstellungsskripte suchen nach einem Quelltextarchive mit dem Namen `<package>_<version>.orig.tar.xz`.

Es gibt eine sehr strenge Regel bei der Debian-Paketerstellung: jegliche Änderung am Quelltextarchiv benötigt ***eine neue Upstream-Versionsnummer**. Geprüft wird dies über eine Checksumme der tar.xz-Datei.



Die Prüfsumme einer tar.xz-Datei ändert sich auch, wenn man von einem unveränderten Verzeichnis erneut ein Archiv erstellt!

Generell sollte man mit dem Paketerstellen nur dann weitermachen, wenn man mit dem Quelltextarchiv soweit einverstanden ist. Jegliche Release-Fixes upstream sollten *vorher* erledigt werden.



Bevor man Debian-Pakete erstellt, sollte man via manueller Installation in ein temporäres Verzeichnis (siehe [Abschnitt 3.2.1](#)) die Installation und danach das Programm selbst prüfen (Übersetzungen komplett, Datenbanken/Beispiele richtig installiert, Dokumentation up-to-date...).

Die Speicherplatzreduktion durch Bereinigen und Packen ist enorm. Der Aufruf von `du` am Ende jeder Operation zeigt die jeweiligen Speicherplatzeinsparungen:

```
196,0M MasterSim-git/
# nach Entfernen von .git
62,0M mastersim-0.9.2/
# nach Bereinigen
9,1M mastersim-0.9.2/
# komprimiert
2,9M mastersim_0.9.2.orig.tar.xz
```

4.4.2. Vorbereiten des Debian-Pakets

Je nach Distributions-Release gibt es kleine Unterschiede in den Konfigurationsdateien. Daher werden diese in individuellen Unterverzeichnissen abgelegt (und im Versionskontrollsystem versioniert).

Es gibt verschiedene Möglichkeiten, die benötigten Konfigurationsdateien zu erstellen. Zum Beispiel kann man `dh_make` nutzen, aber für viele Programme ist das nicht nötig. Daher wird auf die Verwendung des Tools verzichtet und die Dateien selbst generiert.

Es wird nun ein Distro-Release-Verzeichnis erstellt, beispielsweise `ubuntu-20.04-focal` wie oben in der Verzeichnisstruktur gezeigt ist.

In diesem Arbeitsverzeichnis wird ein Unterverzeichnis `debian` erstellt.

4.4.3. Datei `debian/changelog`

Im Distro-Verzeichnis führt man nun den Befehl:

```
dch --create
```

aus. Dies erstellt eine Vorlage für die changelog-Datei und öffnet einen Editor. Man sollte diese Changelog-Datei nun bearbeiten, sodass sie ungefähr so aussieht.

debian/changelog Datei für 20.04 Focal

```
mastersim (0.9.2-1~focal~ppa1) focal; urgency=medium

* Created first package for Focal.

-- Andreas Nicolai <andreas.nicolai@gmx.net> Tue, 04 Jan 2022 17:58:44 +0100
```



Das `dch`-Tool verwendet die vorab gesetzten Umgebungsvariablen für die Unterschriftenzeile, siehe [Abschnitt 4.1.5](#).

Der Changelog-Text beinhaltet ausschließlich Änderungsinformationen über die Paketerstellung, **nicht** über die Upstream-Quelltextänderungen. Diese sind im Quelltextarchiv bzw. in Release-Informationen der Software selbst dokumentiert.

Die erste Changelog-Zeile selbst hat auch ein festgelegtes Format

```
mastersim (0.9.2-1~focal~ppa1) focal; urgency=medium
```

bestehend aus

- Paketnamen
- Paket-Version (siehe [Abschnitt 4.4.4](#))
- Distributions-ID
- Dringlichkeit; hier sollte bei Anwendungspaketen eigentlich *medium* immer ausreichen



Man könnte zwar auch mehrere Distributions-IDs angeben, aber dann wird das Paket von Launchpad als fehlerhaft zurückgewiesen. Es ist grundsätzlich sinnvoller, für jede unterstützte Distributionsversion die Pakete separat zu erstellen und auf der jeweiligen Distro selbst zu testen (geht am besten in einer Virtuellen Maschine).

4.4.4. Versionsnummervergabe

Die Wahl der Versionsnummer muss bestimmten Regeln folgen. Sowohl Launchpad als auch der Debian-Paket-Installer **apt** verwenden die Versionsnummer, um jeweils das aktuellste Paket zu finden und zu installieren. Dabei soll folgende Regel gelten:

- Pakete im Paketrepository werden installiert, sofern sie neuer als die in den Ubuntu-Archiven oder anderen Paketrepos sind
- wird ein Paket von einem Paketrepository in das offizielle Ubuntu-Repo übernommen, so soll es Vorang vor der Paketrepovariante haben

Grundsätzlich funktioniert das bereits gut über die Versionierung mit folgendem Schema:

`<upstream_version>-<deb-paketversion>~<distro>~ppa1`

Also:

```
0.9.2-1~focal~ppa1 - erstes Release für 0.9.2
0.9.2-2~focal~ppa1 - zweites Release für 0.9.2 (Paketkorrektur)
```

Die Paketversion wird immer dann erhöht, wenn - bei gleichem Upstream-Quelltext - das Paket neu erstellt wird. Dies kann notwendig werden, wenn Abhängigkeiten nicht erfüllt sind, oder Launchpad das Paket aus anderen Fehlergründen zurückweist. Dazu mehr in [Abschnitt 5](#).

4.4.5. Datei **debian/control**

Diese Datei enthält die wesentlichen Metadaten des Pakets und muss zumindest hinsichtlich der Abhängigkeiten an das jeweilige Distro-Release angepasst werden.

Beispiel 3. debian/control für 20.04 Focal

```
Source: mastersim
Section: science
Priority: optional
Maintainer: Andreas Nicolai <andreas.nicolai@gmx.net>
Build-Depends: debhelper-compat (= 12), cmake, zlib1g-dev, qtbase5-dev, libqt5core5a, libqt5concurrent5,
libqt5gui5, libqt5network5, libqt5svg5-dev, libqt5xml5, libqt5widgets5
Standards-Version: 4.5.0
Homepage: https://bauklimatik-dresden.de/mastersim
Vcs-Git: https://github.com/ghorwin/MasterSim.git
Vcs-Browser: https://github.com/ghorwin/MasterSim
Rules-Requires-Root: binary-targets

Package: mastersim
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: FMI Co-Simulation Masterprogramm
  MasterSim is an FMI Co-Simulation master and programming library. It
  supports the Functional Mock-Up Interface for Co-Simulation in Version
  1.0 and 2.0. Using the functionality of version 2.0, it implements
  various iteration algorithms that rollback FMU slaves and increase
```

stability of coupled simulations.

Die **control** kann sehr viele Daten enthalten. Im Beispiel oben sind nur die Wichtigsten gezeigt (siehe auch <https://www.debian.org/doc/debian-policy/ch-controlfields.html>).

Die Zeile **Build-Depends** enthält alle Pakete, die zur Erstellung notwendig sind.



Das Paket **build-essential** darf nicht als Abhängigkeit gelistet sein, da es ohnehin eine Grundabhängigkeit ist.

Man darf keine Meta-Pakete (wie z.B. **qt5default**) auflisten, sondern muss alle Abhängigkeiten einzeln benennen.

Grundsätzlich schadet es nicht, mehr Abhängigkeiten aufzulisten als notwendig. Aber es ist effizienter, nur die wirklich benötigten Pakete zu wählen.



Ein einfacher Weg, alle Erstellungsabhängigkeiten zu finden, ist eine VM mit der gewählten Distro zu erstellen. Dann sollte man den Quelltext dorthin kopieren und versuchen zu kompilieren. Fehlende Bibliotheken sind in Paketen enthalten (kann man via google in Paketlisten suchen) und so installiert man nach und nach alle benötigten Pakete und schreibt diese dann in die **Build-Depends** Zeile.

Die Abhängigkeit **debhelper-compat** verlangt in runden Klammern noch den Kompatibilitätsmodus. Der sollte immer dem aktuellen Kompatibilitätsmodus der jeweiligen Distribution entsprechen. Bei Ubuntu 18.04 ist das 10, ab 20.04 ist das 12. Ab 22.04 wird es voraussichtlich 13 sein.

Ebenso muss die **Standards-Version** jeweils angepasst werden, wenn man Warnungen bei der Paketerstellung vermeiden will. Bei 18.04 ist das 4.4.1, bei 20.04 ist das 4.5.0 und bei 21.10 ist das 4.5.1.

Die Laufzeitabhängigkeiten müssen zumeist nicht explizit gegeben werden. Das Feld **Depends** enthält normalerweise den Platzhalter `${shlibs:Depends}` der automatisch bestimmt wird. Beim Bauen des Quellpakets wird beim Linkprozess eine Liste von gelinkten Laufzeitbibliotheken erstellt und die jeweiligen Pakete werden dann als Abhängigkeiten aufgeführt.

Alle weiteren Informationen kann man in der Ubuntu Doku <https://packaging.ubuntu.com/html/debian-dir-overview.html#the-control-file> oder im Debian-Handbuch <https://www.debian.org/doc/debian-policy/ch-source.html> nachlesen.

4.4.6. Datei **debian/copyrights**

Diese Datei enthält Informationen zu den verschiedenen Lizenzen im Quelltextarchiv. Typischerweise haben eingebundene externe Bibliotheken anderen Lizenzen als die eigentliche Anwendung. Dies kann durch Angabe von Unterverzeichnissen und Wildcards definiert werden.

debian/copyrights-Datei

```

Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: mastersim
Upstream-Contact: Andreas Nicolai <andreas.nicolai@gmx.net>
Source: https://github.com/ghorwin/MasterSim

Files: *
Copyright: 2017-2022 Andreas Nicolai
License: BSD-3-clause

Files: debian/*
Copyright: 2021 Andreas Nicolai
License: GPL-2+

License: GPL-2+
This package is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
...

License: BSD-3-clause
Copyright (c) 2022 Andreas Nicolai. All rights reserved.
.
Redistribution and use in source and binary forms, with or without
...
```

Entsprechend dieses Formats lassen sich nun für alle relevanten Teile des Programms individuelle Lizenzen definieren. Die Webseite <https://dep-team.pages.debian.net/deps/dep5/#license-specification> definiert eine Liste von Schlüsselworten, die für die **License** Felder zu verwenden sind.

Weitere Informationen in <https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/>

4.4.7. Datei *debian/rules*

Diese Datei gibt letztlich an, was vor, beim und nach dem Erstellen des Binärpakets alles zu machen ist. Je nach Pakettyp kann das eine Menge sein. Allerdings brauchen wir das (fast) alles nicht, da unser **CMakeLists.txt** Skript bereits die ganze Arbeit macht. Entsprechend ist diese Datei minimalistisch:

debian/rules Datei

```

#!/usr/bin/make -f
# See debhelper(7) (uncomment to enable)
# output every command that modifies files on the build system.
# export DH_VERBOSE = 1

%:
    dh $@
```

Alle oben genannten Dateien sollten im Unterverzeichnis **debian** der Distro-Verzeichnis abgelegt werden.

4.5. Quell-Paket bauen

Um das Quellpaket zu bauen, benötigt man nun eine lokale Kopie des Quelltextverzeichnis. In dieses

wird das **debian** Verzeichnis kopiert.

```
# Im Distro-Arbeitsverzeichnis
$ rm -rf mastersim-0.9.2
$ cp -R ../mastersim-0.9.2/ .
$ cp -R debian/ mastersim-0.9.2/
```

Sind nun alle Dateien abgelegt, wechselt man in das Paketverzeichnis und führt das Paketerstellungstool aus:

```
$ cd mastersim-0.9.2 &&
$ dpkg-buildpackage -S
```

Ausgabe:

```
dpkg-buildpackage: Information: Quellpaket mastersim
dpkg-buildpackage: Information: Quellversion 0.9.2-1~focal~ppa1
dpkg-buildpackage: Information: Quelldistribution focal
dpkg-buildpackage: Information: Quelle geändert durch Andreas Nicolai <andreas.nicolai@gmx.net>
dpkg-source --before-build .
fakeroot debian/rules clean
dh clean
dh_clean
dpkg-source -b .
dpkg-source: Information: Quellformat »3.0 (quilt)« wird verwendet
dpkg-source: Information: mastersim wird unter Benutzung des existierenden ./mastersim_0.9.2.orig.tar.xz gebaut
dpkg-source: Information: mastersim wird in mastersim_0.9.2-1~focal~ppa1.debian.tar.xz gebaut
dpkg-source: Information: mastersim wird in mastersim_0.9.2-1~focal~ppa1.dsc gebaut
dpkg-genbuildinfo --build=source
dpkg-genchanges --build=source >./mastersim_0.9.2-1~focal~ppa1_source.changes
dpkg-genchanges: Information: kompletter Quellcode beim Hochladen hinzufügen
dpkg-source --after-build .
dpkg-buildpackage: Information: Alles hochzuladen (Originalquellen enthalten)
signfile mastersim_0.9.2-1~focal~ppa1.dsc

signfile mastersim_0.9.2-1~focal~ppa1_source.buildinfo

signfile mastersim_0.9.2-1~focal~ppa1_source.changes
```

Es werden im übergeordneten Verzeichnis erstellt:

```
# Archiv mit Steuerungsdaten
mastersim_0.9.2-1~focal~ppa1.debian.tar.xz
# Signaturdatei
mastersim_0.9.2-1~focal~ppa1.dsc
# Build-Informationen
mastersim_0.9.2-1~focal~ppa1_source.build
mastersim_0.9.2-1~focal~ppa1_source.buildinfo
# Hochlade Informationen
mastersim_0.9.2-1~focal~ppa1_source.changes
```

4.6. Quell-Paket prüfen

Das Quellpaket sollte nun auf Fehler überprüft werden (das passiert beim Hochladen auf Launchpad automatisch):

```
# im Distro-Verzeichnis
lintian -EvI --pedantic --show-overrides --color=auto mastersim_0.9.2-1~focal~ppa1_source.changes
```

Ausgabe:

```
N: Using profile ubuntu/main.
N: Starting on group mastersim/0.9.2-1~focal~ppa1
N: Unpacking packages in group mastersim/0.9.2-1~focal~ppa1
N: Finished processing group mastersim/0.9.2-1~focal~ppa1
N: ----
N: Processing changes file mastersim
N: (version 0.9.2-1~focal~ppa1, arch source) ...
N: ----
N: Processing source package mastersim
N: (version 0.9.2-1~focal~ppa1, arch source) ...
N: ----
N: Processing buildinfo package mastersim
N: (version 0.9.2-1~focal~ppa1, arch source) ...
I: mastersim source: debian-watch-file-is-missing
I: mastersim source: testsuite-autopkgtest-missing
X: mastersim source: upstream-metadata-file-is-missing
```

Die Webseite <https://lintian.debian.org/levels> listet die verschiedenen Fehlerstufen auf. Bei Fehlern ("E:") kann das Paket nicht hochgeladen werden und muss erstmal repariert werden.



Fehler und Warnungen werden in der Konsole als Hyperlink angezeigt, welche mit STRG+Klick im Webbrowser geöffnet werden können. Dort gibt es dann eine mehr oder weniger konkrete Hilfestellung.



Versucht man ein fehlerbehaftetes Paket auf Launchpad hochzuladen, wird es normalerweise zurückgewiesen. In diesem Fall kann man aber nicht einfach den Fehler beheben und das Paket erneut hochladen - man muss stattdessen **immer eine neue Paketversion definieren!**. Deshalb lohnt es sich, Pakete mit **lintian** komplett durchzuchecken.

4.7. Binärpaket erstellen und prüfen

Eigentlich analog zur Erstellung und Prüfung des Quellpakets:

```
# Im Paketverzeichnis
$ dpkg-buildpackage
```

Dabei wird *MasterSim* komplett aus dem Quelltext erstellt und sollte natürlich fehlerfrei bauen.

In diesem Schritt werden die folgenden zusätzlichen Dateien im Distro-Verzeichnis erstellt:

```
mastersim_0.9.2-1~focal~ppa1_amd64.buildinfo
mastersim_0.9.2-1~focal~ppa1_amd64.changes
mastersim_0.9.2-1~focal~ppa1_amd64.deb
mastersim-dbgsym_0.9.2-1~focal~ppa1_amd64.ddeb
```

Das Binärpaket kann ebenso mit **lintian** getestet werden:

```
# im Distro-Verzeichnis
# Beim Aufruf Suffix '_amd64' statt '_source' verwenden!
lintian -EvI --pedantic --show-overrides --color=auto mastersim_0.9.2-1~focal~ppa1_amd64.changes
```

Ausgabe:

```
N: Using profile ubuntu/main.
N: Starting on group mastersim/0.9.2-1~focal~ppa1
N: Unpacking packages in group mastersim/0.9.2-1~focal~ppa1
N: Finished processing group mastersim/0.9.2-1~focal~ppa1
N: ----
N: Processing changes file mastersim
N: (version 0.9.2-1~focal~ppa1, arch source amd64) ...
N: ----
N: Processing source package mastersim
N: (version 0.9.2-1~focal~ppa1, arch source) ...
N: ----
N: Processing buildinfo package mastersim
N: (version 0.9.2-1~focal~ppa1, arch amd64 source) ...
N: ----
N: Processing binary package mastersim
N: (version 0.9.2-1~focal~ppa1, arch amd64) ...
N: ----
N: Processing binary package mastersim-dbgsym
N: (version 0.9.2-1~focal~ppa1, arch amd64) ...
E: mastersim: embedded-library usr/bin/MasterSimulator: tinyxml
E: mastersim: embedded-library usr/bin/MasterSimulatorUI: tinyxml
I: mastersim source: debian-watch-file-is-missing
I: mastersim source: testsuite-autopkgtest-missing
P: mastersim source: package-does-not-install-examples examples/
X: mastersim source: upstream-metadata-file-is-missing
```

Interessant ist hier die Fehlermeldung über die eingebettete externe Bibliothek. Das Prüftool erkennt anhand der Linker-Eingabedateien, dass MasterSimulator und MasterSimulatorUI gegen Symbole linken, die ihrerseits bereits in einer bereits veröffentlichten Bibliothek **tinyxml** vorhanden sind. Das sollte man ja, wie eingangs erläutert, vermeiden. Diese Fehlermeldung kann aber ignoriert werden (geht im Fall von MasterSim auch gar nicht anders, da **tinyxml** selbst nicht mehr weiterentwickelt wird). Das Binärpaket wird trotzdem so auf Launchpad akzeptiert.

4.8. Veröffentlichung auf dem Launchpad PPA

Abschließend bleibt nur noch die Veröffentlichung auf dem eigenen PPA auf Launchpad. Dafür gibt es ebenfalls ein kleines Tool **dput**:

```
$ dput ppa:ghorwin/sim ../mastersim_0.9.2-1~focal~ppa1_source.changes
```

Allerdings führt dies zunächst zu einer Fehlermeldung:

```
Checking signature on .dsc
gpg: /home/ghorwin/svn/MasterSim_trunk/debian/ubuntu-20.04-focal/mastersim_0.9.2-1~focal~ppa1.dsc: Valid signature from
6E0814BD3FCA8338
Checksum doesn't match for /home/ghorwin/svn/MasterSim_trunk/debian/ubuntu-20.04-focal/mastersim_0.9.2-1~focal~ppa1.dsc
```

Denn beim Erstellen des Binärpakets wurde die Prüfsummendatei **.dsc** verändert. Daher muss man noch einmal ein Source-Paket erstellen:

```
# im Paketverzeichnis
$ dpkg-buildpackage -S
```

und danach klappt es auch mit dem Hochladen.

5. Aktualisieren von Open-Source Quellpaketen

Es gibt letztlich 2 Gründe, warum man Pakete aktualisieren muss:

- der Upstream-Quelltext hat sich verändert
- das Paket funktioniert nicht; entweder es gab Fehler beim Hochladen (Tests sind fehlgeschlagen, Projekt hat fehlende Abhängigkeiten, etc.) oder durch Aktualisierung des Distro-Releases (bspw. Wechsel von 20.04.3 auf 20.04.4) haben sich in abhängigen Paketen Änderungen ergeben, weswegen man das Paket neu konfigurieren muss

5.1. Neues Upstream-Release

In diesem Fall muss man den in [Abschnitt 4.4](#) beschriebenen Prozess nochmal komplett neu durchlaufen. Da die **debian/*** Dateien schon existieren, muss man hier lediglich prüfen, ob sich was geändert hat. Im Wesentlichen muss die **debian/changelog**-Datei erweitert werden.

Dazu kann man wieder das Tool **dch** aufrufen, wobei man gleich auf der Kommandozeile die neue Versionsnummer angeben kann.

```
# im Distro-Release-Verzeichnis
$ dch -v 0.9.3-1~focal~ppa1
```

dch setzt den neuen Paket-Versionseintrag zunächst auf die Distribution **UNRELEASED**. Entweder man ändert das manuell auf die geforderte Ubuntu-Distro, oder ruft vor dem Quellpaketbauen noch

```
# im Distro-Release-Verzeichnis  
$ dch --release
```

auf. Im weiteren Verlauf (wie oben beschrieben) wird dann das neue Quell-Archiv nebst Quelltexttarball hochgeladen und steht kurz nachher zur Verfügung.

5.2. Paketaktualisierung

Soll nur das Paket selbst aktualisiert werden, muss man letztlich nur die **debian/changelog** anpassen, wobei die Upstream-Versionsnummer unverändert bleibt. Man ändert die Version also z.B. von:

0.9.2-1~focal~ppa1 → **0.9.2-2~focal~ppa1**

Beim Hochladen des Pakets wird erwartet, dass sich der Quelltext nicht verändert hat. Sollte man z.B. die **.orig.tar.xz**-Datei neu komprimiert haben, oder der Quelltext im Quelltextverzeichnis hat sich anderweitig verändert, wird die Veröffentlichung des Pakets mit einer Fehlermeldung zurückgewiesen:



```
Rejected:  
File <UPLOADED_FILE> already exists in <LOCATION>, but uploaded version has different contents.  
See more information about this error in https://help.launchpad.net/Packaging/UploadErrors.
```

In diesem Fall sollte man vom Paketrepository die bereits hochgeladene **.orig.tar.xz**-Datei herunterladen, ins Basisverzeichnis kopieren, und dort neu entpacken. Dann ist der Zustand wieder genau hergestellt, und man kann die Paketaktualisierung durchführen.

6. Veröffentlichung als Binärpaket auf eigenem Server

6.1. Paketerstellung mit CPack

Eigene Binärpakete lassen sich bei bereits vorbereiteter **CMakeLists.txt**-Datei (siehe [Abschnitt 3.2](#)) am einfachsten mit CPack erstellen.

Dazu werden in der **CMakeLists.txt**-Datei einfach die notwendigen CPACK-Variablen gesetzt, siehe offizielle Dokumentation: <https://cmake.org/cmake/help/latest/module/CPack.html> und speziell für Debian Pakete: https://cmake.org/cmake/help/latest/cpack_gen/deb.html

Ein detailliertes Tutorial <https://decovar.dev/blog/2021/09/23/cmake-cpack-package-deb-apt/> beschreibt die Konfiguration der **CMakeLists.txt** im Detail.

Zum Abschluss (nach dem `cmake` Aufruf) folgt die Paketerstellung mit:

```
$ cpack -G DEB
```



Die Variable `CPACK_DEBIAN_PACKAGE_DEPENDS` ist wichtig, da nur damit eine reibungslose Installation möglich ist und automatisch alle abhängigen Pakete mit installiert werden. Und das ist letztlich ja der Sinn der deb-Erstellung, sonst reicht das Stand-Alone Anwendungsarchiv *.7z ja aus.

6.2. Hosting der Pakete in eigenem Repository

Eine detaillierte Anleitung findet sich hier: <https://earthly.dev/blog/creating-and-hosting-your-own-deb-packages-and-apt-repo/>

Das Ganze beinhaltet eine Reihe von Aufgaben, die man am besten via Skript erledigt.

- Verzeichnisstruktur erstellen
- Signaturschlüssel erstellen und hochladen
- dann die Verzeichnisstruktur befüllen (deb-Pakete hochladen)
- Liste der Pakete aktualisieren
- hash-codes erzeugen
- Veröffentlichung signieren
- alles hochladen