Bachelorarbeit

im Studiengang

Informatik

zur Erlangung des Grades eines

Bachelor of Science

Thema

Analyse und Anwendung moderner Paketmanagersysteme für Linux

Vorgelegt von:

Kejvi Hysenbelli

geb. am 21.07.2000 in Poliçan, Albanien
Eckenheimer Landstr. 178, 60319 Frankfurt am Main

Matrikel-Nr.: 1399338

im Winter-Semester 24/25
am Fachbereich 2: Informatik und Ingenieurwissenschaften
der Frankfurt University of Applied Sciences

****

Erstprüfer: Prof. Dr. Christian Baun

Zweitprüfer: Prof. Dr. Thomas Gabel

Abgabedatum: 17.12.2024

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Unterschrieben: Kejvi HYSENBELLI, 1399338, 16. Dezember 2024

Vorwort

Diese Arbeit habe ich im Rahmen meines Bachelorstudiums an der Frankfurt University of Applied Sciences zur Erlangung des akademischen Grades Bachelor of Sciences verfasst.

Den praktischen Teil dieser Bachelorarbeit habe ich selbständig entwickelt um den Hauptprinzip näher zu kommen. Für die Unterstützung und Beratung seitens der Fachhochschule Frankfurt möchte ich mich bei Prof. Dr. Christian Baun und Prof. Dr. Thomas Gabel bedanken.

Nicht zuletzt möchte ich mich bei meiner Familie und meinen Freunden für ihre Unterstützung während dieser Zeit bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Vielzahl an Linux-Distributionen hat zu Herausforderungen hinsichtlich der Softwarepaketierung geführt. Traditionelle Systeme wie .deb und .rpm verlangen von den Entwicklern, dass sie mehrere Pakete für verschiedene Distributionen pflegen, was zu Kompatibilitäts- und Wartungsproblemen führt.

Mit der zunehmenden Popularität des Betriebssystems Linux haben sich universell einsetzbare Paketierungslösungen – sogenannte AppImages, Flatpak und Snapcraft – entwickelt, welche die Bereitstellung vereinfachen, die Portabilität erhöhen und die Sicherheit verbessern sollen.

Jedes System verfolgt einen anderen Ansatz: AppImage fokussiert sich auf Einfachheit und Unabhängigkeit von Distributionen. Flatpak legt den Schwerpunkt auf Sandboxing und Sicherheit, und Snapcraft ist für sein umfassendes Ökosystem und die Integration in Ubuntu bekannt.

1.2 Aufgabenstellung

Die vorliegende Arbeit befasst sich mit der Analyse und Anwendung moderner Linux-Paketmanagementsysteme, insbesondere AppImage, Flatpak und Snap. Ziel ist die systematische Untersuchung der Unterschiede, Stärken und Schwächen dieser Ansätze, um ein tiefgehendes Verständnis der praktischen und technischen Aspekte zu gewinnen.

Im Rahmen der Untersuchung wird die GNOME-Anwendung 'Dia' als Fallstudie herangezogen. Die vorliegende Software dient als Fallstudie, anhand derer die drei Verpackungsmethoden konkret implementiert und hinsichtlich ihrer Effizienz sowie Benutzerfreundlichkeit bewertet werden.

Ein besonderes Augenmerk gilt der Analyse wesentlicher Merkmale, zu denen unter anderem das Abhängigkeitsmanagement, das Sandboxing, die Sicherheit, die Portabilität sowie der Ressourcenverbrauch zählen. Der praktische Verpackungsprozess ermöglicht nicht nur die Gewinnung theoretischer Einsichten, sondern auch die Aneignung praktischer Erfahrungen, die für Entwickler von Relevanz sind.

In der Schlussbetrachtung erfolgt ein Vergleich der Systeme sowie die Formulierung von Empfehlungen zu deren optimaler Nutzung. Dabei werden die zukünftigen Implikationen dieser Paketmanagementlösungen für das Linux-Ökosystem, insbesondere hinsichtlich ihrer Rolle bei der Verteilung und Installation von Software, herausgearbeitet. Ziel der Arbeit ist es, einen Beitrag zur besseren Verständlichkeit und Nutzung moderner Linux-Paketierungsmethoden zu leisten.

1.3 Vorgehensweise

Die vorliegende Arbeit ist in mehrere Schritte gegliedert, welche sich mit der Analyse und Anwendung moderner Linux-Paketmanagementsysteme beschäftigen. Im Kern der Untersuchung stehen die drei Paketierungsansätze AppImage, Flatpak und Snap, welche anhand der Software 'Dia' beleuchtet und miteinander verglichen werden. Ziel ist es, die verschiedenen Aspekte dieser Systeme zu beleuchten und gegenüberzustellen.

1.3.1 Vorbereitung und Auswahl der Fallstudie

Die GNOME-Anwendung 'Dia', ein Diagrammeditor, dient als Fallstudie. Zunächst wird der Quellcode der Anwendung aus einem Repository bezogen, notwendige Abhängigkeiten installiert und die Anwendung für die anschließende Paketierung vorbereitet.

1.3.2 Detaillierte Verpackung in drei Formate

AppImage: Hier wird die Anwendung in ein eigenständiges, ausführbares Paket transformiert. Dabei liegt der Fokus auf der Portabilität und Einfachheit der Erstellung.

Flatpak: Dieses Format nutzt eine Laufzeitumgebung und Sandbox-Technologie, um Sicherheit und Konsistenz zu gewährleisten. Der Prozess umfasst die Erstellung einer YAML-Manifestdatei, die Konfiguration und den Aufbau des Pakets.

Snap: Durch Bündelung aller notwendigen Abhängigkeiten wird ein in sich geschlossenes Paket geschaffen. Die YAML-Datei definiert die spezifischen Anforderungen und Berechtigungen.

1.3.3 Vergleich der Paketierungsansätze

Es werden zentrale Kriterien wie Benutzerfreundlichkeit, Speicherbedarf, Sicherheit und Ressourceneffizienz untersucht. Hierbei kommen Tests zur Anwendung, die die praktische Nutzung der Pakete beleuchten.

1.3.4 Zusammenfassung und Bewertung

Die Ergebnisse der Analyse werden zusammengeführt, um die Vor- und Nachteile der einzelnen Systeme darzustellen. Abschließend wird die zukünftige Rolle dieser Technologien für Linux bewertet.

Die strukturierte Vorgehensweise gewährleistet, dass die Untersuchung der modernen Paketmanagementsysteme sowohl theoretisch fundiert als auch praktisch relevant ist.

1.4 Aufbau der Arbeit

1.4.1 Kapitel 2

Dieses Kapitel umfasst die Entwicklung und Geschichte der Linux-Paketverwaltungssysteme.

1.4.2 Kapitel 3

Dieses Kapitel beschreibt die verschiedenen Verpackungsprozesse, die in dieser Arbeit Thema sind, und gibt einen detaillierten Einblick in den Verpackungsweg mit der Anwendung Dia.

1.4.3 Kapitel 4

Ein ausführlicher Vergleich zwischen den angewandten Paketmanagementsystemen mit einem Einblick in Speicherbedarf, Vorteile, Nachteile und Komplexität.

1.4.4 Kapitel 5

Der Abschluss der Arbeit, der einen Einblick in die Gedanken, den aufgetretenen Herausforderungen und einen Ausblick auf die Zukunft gibt.

Kapitel 2

Entwicklung von Linux-Paketverwaltungssystemen

2.1 Grundsätze der Linux-Paketverwaltung

Die Installation von Software unter Windows erfolgt typischerweise durch einen Doppelklick auf ein Installationsprogramm. Dieses sorgt dafür, dass alle relevanten Dateien des Programms automatisch an den vorgesehenen Speicherorten abgelegt werden. Im Vergleich dazu weisen Linux-Pakete einige spezifische Merkmale auf:[1]

- Ein Linux-Paket besteht aus einer einzigen Datei, die entweder lokal auf einem Datenträger gespeichert oder über das Internet übertragen werden kann.
- Im Gegensatz zu Windows-Installationsprogrammen handelt es sich bei Linux-Paketdateien nicht um eigenständige Programme. Für die Installation benötigen sie spezielle Software, die den Installationsvorgang übernimmt.
- Jedes Paket enthält Metadaten, welche Informationen zur Version bereitstellen. Diese ermöglichen es der Paketverwaltungssoftware, die neueste Version eines Pakets zu identifizieren.
- Die Architekturinformationen in einem Paket spezifizieren, für welchen CPU-Typ die Software ausgelegt ist.

2.1.1 Ehemalige Paketverwaltungssysteme

In der Vergangenheit erfolgte die Arbeit mit Paketen auf lokaler Ebene. Dies implizierte, dass zur Installation eines Pakets auf einem Computer zunächst eine Paketdatei aus dem Internet oder auf andere Weise heruntergeladen werden musste. Erst nach Ausführung eines lokalen Befehls konnte die Installation des Pakets erfolgen.

Der zuvor beschriebene Ansatz kann jedoch als mühsam empfunden werden, wenn ein Paket zahlreiche Abhängigkeiten aufweist. Dies kann dazu führen, dass zunächst eine Installation versucht wird, wobei unerfüllte Abhängigkeiten festgestellt werden. In der Folge werden mehrere weitere Pakete heruntergeladen, von denen ebenfalls einige unerfüllte Abhängigkeiten aufweisen. Dieser Prozess kann sich wiederholen, bis

alle Abhängigkeiten erfüllt sind. Bis alle abhängigen Pakete identifiziert sind, müssen unter Umständen ein Dutzend oder mehr Pakete installiert werden.[1]

Dies verdeutlicht, wie zeitaufwändig und fehleranfällig das manuelle Verwalten von Paketabhängigkeiten sein kann, was die Notwendigkeit moderner Paketmanager unterstreicht.

2.1.2 Die heutigen Paketverwaltungssysteme

Heutzutage sind Paketverwaltungssysteme ein Muss für die Verwaltung von Linux-Umgebungen. Die Normalisierung der Art und Weise, wie Software installiert, aktualisiert und Abhängigkeiten aufgelöst werden, erleichtert den Endbenutzern die Wartung ihrer Infrastruktur.

Sie stellen eine Verbindung zu vielen zentralisierten Repositorien her, so dass die Benutzer auf eine breite Palette von Software zugreifen können, wobei die Abhängigkeiten automatisch gehandhabt werden, was die Arbeit erleichtert und Konflikte reduziert. Weniger Aufwand für den Benutzer, mehr Stabilität für das System (Paketmanager sorgen dafür, dass die Installationen konsistent sind).

Diese Systeme ermöglichen eine besser organisierte Verteilung von Anwendungen. Allerdings sind sie oft auf eine bestimmte Linux-Distribution beschränkt und erfordern mehr Arbeit, um die Software für andere Umgebungen zu paketieren. Dies wird zu einem Problem, wenn Sie sich an ein allgemeines Linux-Publikum wenden.[2]

2.1.3 Moderne Paketverwaltungssysteme

Es lässt sich konstatieren, dass moderne Paketmanager die Softwareverteilung wirklich revolutioniert haben, indem sie den Schwerpunkt auf die Kompatibilität zwischen allen Plattformen legen und Entwicklern die Arbeit erheblich erleichtern. Diese neuen Paketmanager reduzieren die systemspezifische Paketierung auf ein Minimum, da die Anwendungen ihre eigenen Abhängigkeiten bündeln können und die Frameworks selbst mit mehreren Linux-Distributionen sofort einsatzbereit sind. Das bedeutet, dass Entwickler weniger Software für verschiedene Distributionen neu paketieren müssen und sich die Anwendungen in verschiedenen Umgebungen einheitlicher verhalten. Benutzer schätzen die bessere No-Install-Erfahrung moderner Systeme. Anwendungen werden so verpackt, dass sie isoliert vom zugrunde liegenden System und auch eigenständig ausgeführt werden können. Diese Philosophie bietet eine bessere Sicherheit und Stabilität, indem sie Konflikte mit Systembibliotheken minimiert, hat aber auch einige Nachteile, wie die Vergrößerung der Anwendungen.[3]

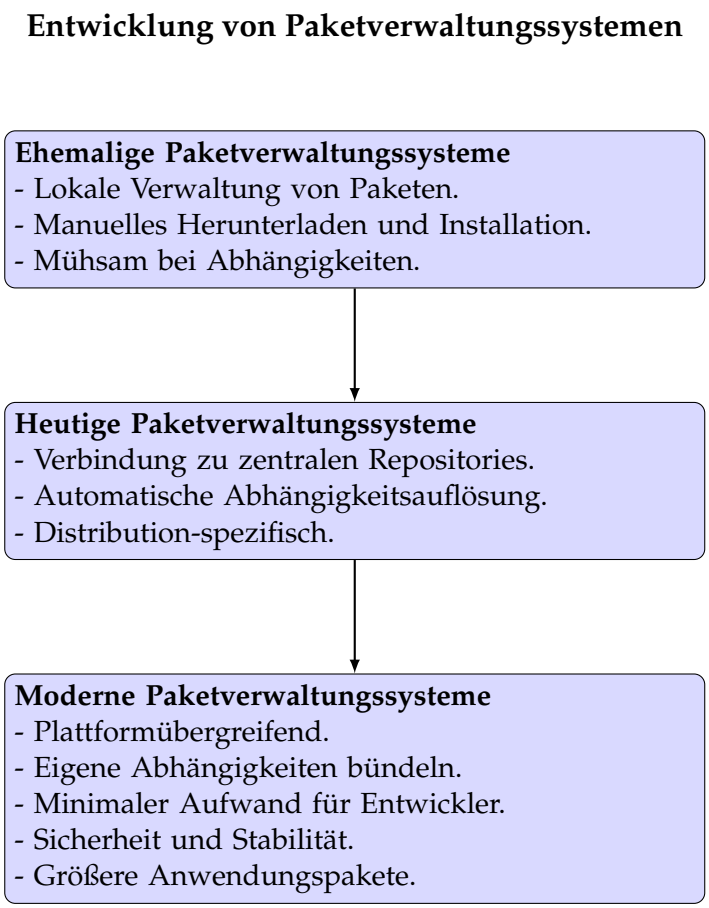

ABBILDUNG 2.1: Entwicklung von Paketverwaltungssystemen

Kapitel 3

Vergleich von Verpackungsprozessen anhand eines Anwendungsbeispiels.

Dieses Kapitel konzentriert sich auf eine detaillierte Anleitung zur Verpackung der Anwendung Dia mittels moderner Linux-Paketmanagement-Tools wie: AppImage, Flatpak, und Snap. Jeder Abschnitt konzentriert sich auf den Erstellungsprozess, wobei Konfigurationsdetails, Herausforderungen und Lösungen hervorgehoben werden.

3.1 Dia aus dem Quellcode kompilieren

Bevor die Dia-Anwendung in moderne Paketformate wie Appimage, Snap oder Flatpak verpackt werden kann, ist die Erstellung der Anwendung aus dem Quellcode ein grundlegender Schritt. Dieser Prozess umfasst das Herunterladen und Kompilieren des Quellcodes sowie das Extrahieren der erzeugten Binärdateien, die als Grundlage für die anschließende Paketierung dienen.

3.1.1 Dia aus dem Gitlab-Repository klonen

Der erste Schritt im Dia-Kompilierungsprozess besteht darin, ein Terminal in dem Zielverzeichnis zu öffnen, in dem die Anwendung erstellt werden soll. Durch die Eingabe der folgenden Befehle wird der Klonprozess eingeleitet:

Wir stellen sicher, dass git installiert ist, bevor wir diesen Befehl ausführen

```
$ git clone https://gitlab.gnome.org/GNOME/dia.git
```

3.1.2 Abhängigkeiten installieren

Da Dia mit Meson gebaut wird, ist Meson zu installieren.

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt update
$ sudo apt install meson
```

Laut der Datei *meson.build*, die sich im Stammverzeichnis von Dia befindet, sind die folgenden Abhängigkeiten erforderlich, damit Dia gebaut werden kann.

```
$ sudo apt update && sudo apt install -y libglib2.0-dev \
libgtk-3-dev libxml2-dev zlib1g-dev libgraphene-1.0-dev \
gettext libxslt1-dev xsltproc libemf-dev appstream-util \
cmake libxpm-dev
```

3.1.3 Build Dia

Nachdem alle erforderlichen Abhängigkeiten installiert waren, konnte mit dem Bau von Dia begonnen werden. Gemäß der hier vorliegenden Dia-Dokumentation waren die folgenden Befehle notwendig, um Dia zu bauen:[4]

```
$ cd path/to/dia
$ meson setup build -Dprefix=/build/install
$ ninja -C build
$ ninja -C build install
```

3.2 AppImage

Es wurde 2004 unter dem Namen klik von Simon Peter eingeführt und 2011 in PortableLinuxApp umbenannt. 2013 wurde es schließlich so benannt, wie wir es heute kennen: AppImage. AppImages zeichnen sich durch ihre Kompaktheit und Vielseitigkeit aus, da es sich um einzelne ausführbare Dateien handelt, die alle erforderlichen Komponenten für die Ausführung einer gesamten Anwendung enthalten. AppImages sind einzigartige Binärdateien, die dem Grundprinzip „eine Anwendung - eine Datei“ folgen. Dadurch wird die Verwaltung mehrerer Versionen einer einzigen Anwendung auf einem bestimmten System vereinfacht. Für diejenigen, die mit Windows arbeiten, ist die Ähnlichkeit mit portablen .exe-Dateien auffällig.[5]

Für Anwendungsentwickler bieten AppImages eine Möglichkeit, Linux-Pakete ohne die Komplexität der Paketierung für mehrere Distributionen bereitzustellen. Durch die Verwendung eines einzigen Build-Prozesses gewährleisten AppImages Kompatibilität über verschiedene Linux-Umgebungen hinweg und reduzieren die Notwendigkeit, sich mit den Abhängigkeiten und Kompatibilitätsproblemen jeder einzelnen Distribution zu befassen.[6]

3.2.1 Verpackungsprozess

Es gibt viele verschiedene Ansätze zur Erstellung von AppImages. Oft hängt der richtige Weg, ein AppImage zu verpacken, von der Anwendung ab, die Sie darin unterbringen wollen, der so genannten Nutzlast. Verschiedene Programmiersprachen (oder besser gesagt, verschiedene Anwendungstypen (d. h. native Binärdateien, Skripte, Bytecode usw.)) erfordern unterschiedliche Methoden.[7]

Alle Versuche der Bündelung von Anwendungen haben eines gemeinsam: das „Eingabeformat“, das dann mit appimagetool in ein AppImage umgewandelt wird. Dieses Eingabeformat wird AppDir genannt.

Zur Erstellung des AppDir, das als strukturierte Umgebung für die benötigten Dateien einer Anwendung dient, wurde die Meson-Methode verwendet. Diese Methode ermöglicht eine effiziente und automatisierte Erstellung des Verzeichnisses, das für die anschließende Paketierung unerlässlich ist.

```
$ meson setup build --prefix=$(pwd)/Dia.AppDir/usr
```

Mit diesem Befehl wird Meson so konfiguriert, dass Binärdateien, Bibliotheken und andere Ressourcen im Verzeichnis Dia.AppDir/usr abgelegt werden. Meson wird nun diese Struktur während des Build-Prozesses verwenden.

In der letzten Phase der Erstellung von Dia aus dem Quellcode kommt das Werkzeug Ninja zum Einsatz. Dieses Werkzeug kompiliert die Quellcodedateien effizient und installiert die erzeugten Elemente automatisch in das zuvor erstellte AppDir.

```
$ ninja -C build
$ ninja -C build install
```

Dadurch wird das Verzeichnis Dia.AppDir mit den erforderlichen Binärdateien, Bibliotheken, Symbolen und anderen Ressourcen gefüllt, wie in der Meson-Build-Konfiguration angegeben.

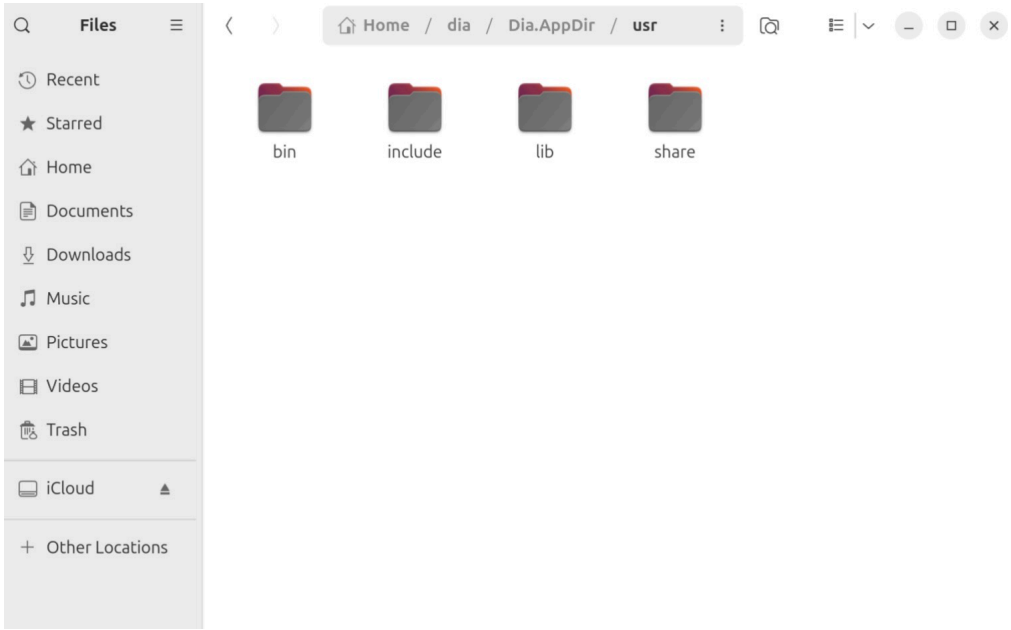

ABBILDUNG 3.1: AppDir Struktur

Die restlichen AppDir-Anforderungen wurden manuell hinzugefügt. Ein

AppImage braucht ein AppRun dass wie ein Einstiegspunkt für den Beginn des Programms funktioniert, ein .desktop datei sowohl auch ein Icon.

Die Desktop-Datei wird automatisch erstellt, wenn Dia kompiliert wird, also kopieren wir sie einfach in das Stammverzeichnis von Dia.AppDir. Der Speicherort ist:

path/to/Dia.AppDir/usr/share/applications/

Das Icon ist auch im Dia-Verzeichnis enthalten. Wir müssen es auch in das Stammverzeichnis von AppDir kopieren. Der Speicherort ist:

path/to/Dia.AppDir/Dia.AppDir/usr/share/icons/hicolor/scalable/apps/

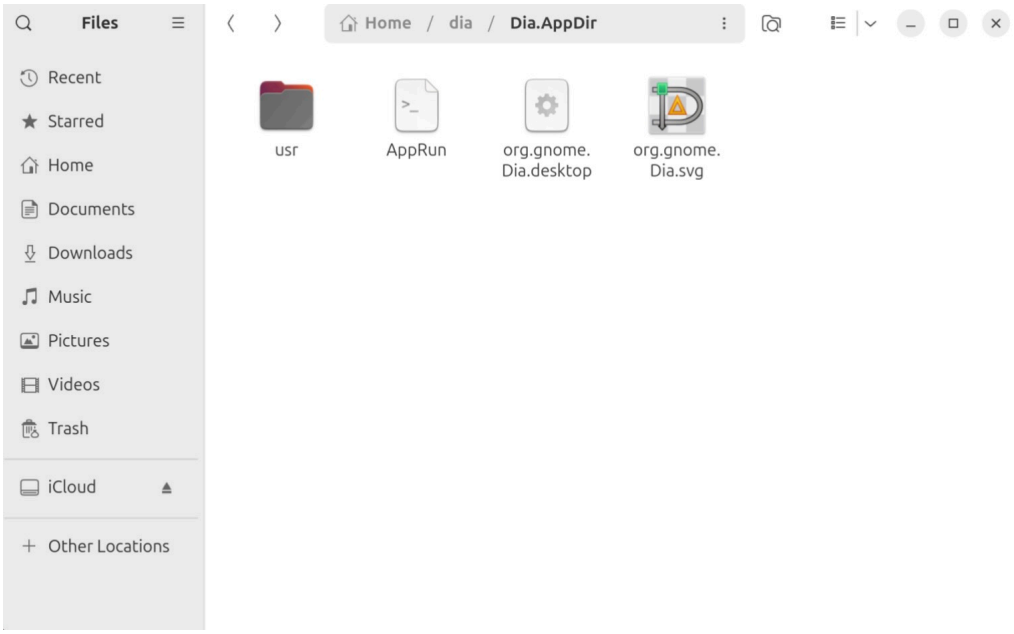

ABBILDUNG 3.2: Kompletes AppDir

Da Dia erwartet, dass sich die ui-Dateien im Pfad Dia.AppDir/usr/share/dia/data befinden, musste eine kleine Änderung in der AppDir-Struktur vorgenommen werden. Der Ordner ui wurde in einen neuen Ordner namens data verschoben.

AppRun definiert Umgebungsvariablen, die von der Anwendung benötigt werden, um Bibliotheken, Datendateien und andere Ressourcen zu finden.

Um ein voll funktionsfähiges AppRun zu erstellen, wird die Datei meson.build von dia benötigt. Sie enthält alle Umgebungsvariablen, die dia für eine erfolgreiche Erstellung und Ausführung benötigt.

Da AppImages in sich vollständig sind und in ein temporäres Verzeichnis

gemountet werden, stellt AppRun sicher, dass die Anwendung ihre Ressourcen relativ zum Mount-Punkt finden kann. Dadurch ist dia unabhängig vom ursprünglichen AppDir und funktioniert in jeder Linux-Distribution.

LISTING 3.1: AppRun

```
#!/bin/bash

# Dynamische Bestimmung des Verzeichnisses, in dem sich das
   Skript selbst befindet
APPDIR="$(dirname "$(readlink -f "$0")")"

# Umgebungsvariablen ueberschreiben
export DIA_BASE_PATH="$APPDIR/usr/share/dia"
export DIA_LIB_PATH="$APPDIR/usr/lib/aarch64-linux-gnu/dia"
export DIA_SHAPE_PATH="$APPDIR/usr/share/dia/shapes"
export DIA_XSLT_PATH="$APPDIR/usr/share/dia/xslt"
export DIA_PYTHON_PATH="$APPDIR/usr/share/dia"
export DIA_SHEET_PATH="$APPDIR/usr/share/dia/sheets"
export DIA_UI_PATH="$APPDIR/usr/share/dia/data/ui"
export LD_LIBRARY_PATH="$APPDIR/usr/lib/aarch64-linux-gnu:\
${LD_LIBRARY_PATH:-}"

# Dia ausfuehren
exec "$APPDIR/usr/bin/dia" "$@"
```

Nachdem das AppDir eingerichtet wurde, wurde das Appimagetool-aarch64 heruntergeladen. Dieses Tool ermöglicht die Erzeugung eines AppImages aus einem bestehenden AppDir.

Das Befehl zum Herunterladen des Appimagetools lautet:

```
$ wget https://github.com/AppImage/AppImageKit/\
    releases/download/continuous/appimagetool-aarch64.AppImage
```

Nach der Installation musste das App-Image-Tool als ausführbar festgelegt werden, um den Paketierungsprozess starten zu können.

```
$ cd path/to/appimagetool-aarch64.AppImage\
$ chmod +x appimagetool-aarch64.AppImage
```

Nach erfolgreicher Konfiguration des AppDir, der Integration aller erforderlichen Komponenten und der Installation und Einstellung des Appimage-Tools als ausführbar, kann die Erstellung des AppImage eingeleitet werden. Dieser Schritt verwandelt die vorbereitete Anwendung in ein eigenständiges, portables Paket, das ohne Installation auf verschiedenen Systemen ausgeführt werden kann.

```
ARCH=aarch64 ./appimagetool-aarch64.AppImage dia/Dia.AppDir
    Dia.AppImage
```

Dieser Befehl verpackt das AppDir in ein Appimage mit Namens Dia.AppImage.

Nach der Erstellung des AppImage wurde festgestellt, dass die erzeugte Datei weder das Symbol der Dia-App anzeigt noch im Anwendungsmenü des Systems erscheint.
Dies ist das Hauptproblem mit dem AppImage-Format. Die Anwendungen werden nicht direkt im System installiert. Daher war ein zusätzliches Tool namens AppImageLauncher erforderlich.[8]

Mit dem folgenden befehl ladet das AppImageLauncher für arm64 systemen unter:

```
$ wget https://github.com/TheAssassin/AppImageLauncher/\
 releases/download/continuous/appimagelauncher_2.2.0-\
 gha111.d9d4c73+bionic_arm64.deb
```

Dann installiert man AppimageLauncher wie folgt:

```
$ sudo dpkg -i /home/parallels/dia/appimagelauncher_2.2.0-\
 gha111.d9d4c73+bionic_arm64.deb
```

Nach der Installation genügt ein Doppelklick auf die Dia.Appimage-Datei, um zum AppimageLauncher zu gelangen, der nach der Erlaubnis fragt, das Appimage zu starten und zu integrieren. Durch einen Klick auf Ausführen und Integrieren wird die Dia.Appimage Datei als normale Desktop-Anwendung in das App-Menü integriert[8].

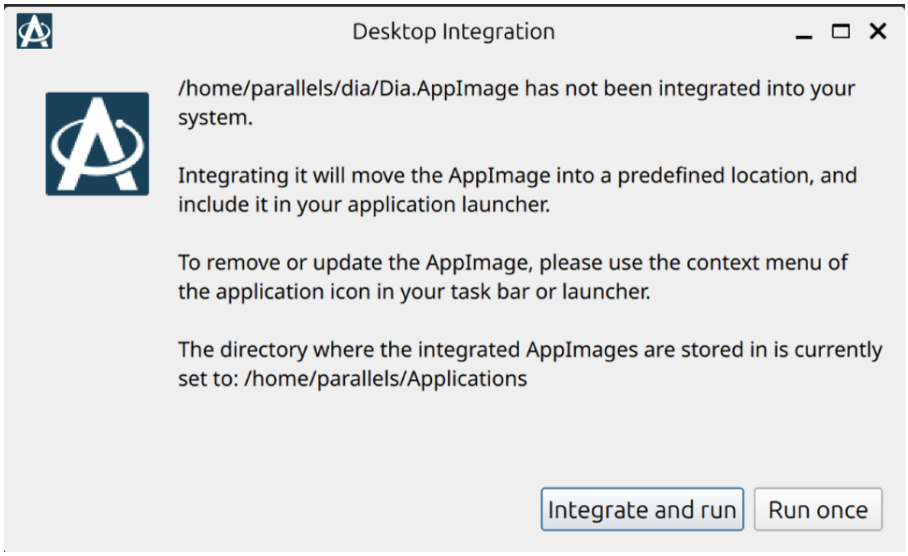

ABBILDUNG 3.3: AppImage Launcher

Nach diesen Schritten war Dia schließlich als Appimage verpackt und funktionierte wie jede andere Anwendung.

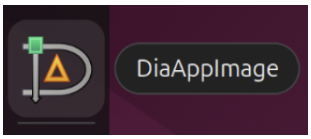

ABBILDUNG 3.4: Dia(AppImage)

3.2.2 Verteilung und Desktop Integration an anderen Linux Distributionen

Die Verteilung einer Appimage-Datei an andere Linux-Distributionen kann mittels eines USB-Sticks oder alternativer Übertragungsmethoden erfolgen. Nach dem erfolgreichen Transfer kann die Anwendung durch einen Doppelklick ausgeführt werden.

Für eine Desktop-Integration ist lediglich die Installation des AppimageLauncher-Programms erforderlich, welches durch einen Doppelklick auf die .Appimage-Anwendung durchgeführt wird.

3.3 Flatpak

Flatpak ist ein System zur Bereitstellung und Verwaltung von Software unter Linux, das Anwendungen in einer isolierten Umgebung ausführt. Dies erhöht die Sicherheit und ermöglicht die Nutzung von Anwendungen unabhängig von der spezifischen Linux-Distribution. Ursprünglich als „xdg-app“ bekannt, wurde es 2016 in Flatpak umbenannt.[9]

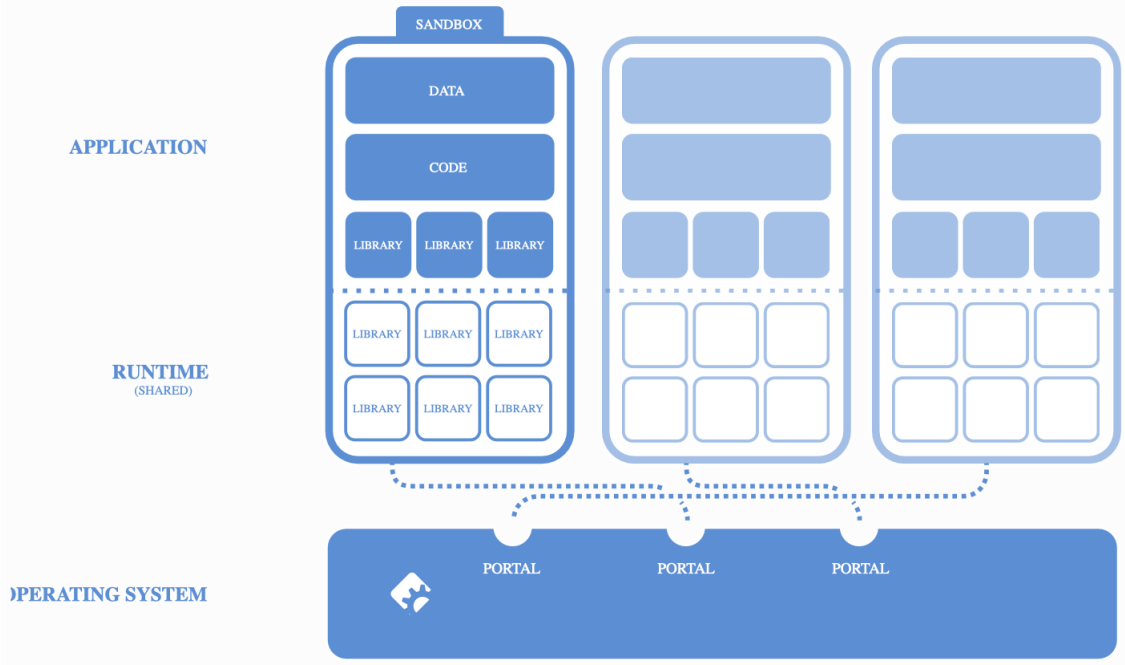

ABBILDUNG 3.5: Flatpak Struktur
[10]

Ein Grundprinzip von Flatpak sind die sogenannten Runtimes. Diese stellen die grundlegenden Abhängigkeiten bereit, die eine Anwendung benötigt, und werden beim Start der Anwendung ins Dateisystem eingebunden. Die verfügbaren Laufzeitumgebungen umfassen eine Vielzahl von Varianten, von minimalistischen, sogenannten Widgets-basierten Laufzeiten bis hin zu umfassenderen Lösungen, welche von Desktop-Umgebungen wie GNOME oder KDE bereitgestellt werden. Anwendungen werden gegen eine spezifische Runtime gebaut, die auf dem Host-System installiert sein muss, damit die Anwendung ausgeführt werden kann.[11]

Falls eine Anwendung weitere Abhängigkeiten benötigt, die nicht in der Runtime enthalten sind, können diese direkt mit der Anwendung gebündelt werden. Dies ermöglicht es, Abhängigkeiten zu nutzen, die in der jeweiligen Distribution nicht verfügbar sind, oder andere Versionen von Abhängigkeiten zu verwenden. Sowohl Runtimes als auch **Anwendungs-Bundles** können pro Benutzer oder systemweit installiert werden.

Falls eine Anwendung weitere Abhängigkeiten benötigt, die nicht in der Runtime enthalten sind, können diese direkt mit der Anwendung gebündelt werden. Dies ermöglicht es, Abhängigkeiten zu nutzen, die in der jeweiligen Distribution nicht verfügbar sind, oder andere Versionen von Abhängigkeiten zu verwenden. Sowohl Runtimes als auch **Anwendungs Bundles** können pro Benutzer oder systemweit installiert werden.

Ein weiteres wesentliches Element von Flatpak ist die Verwendung von Software Development Kits (SDKs). Ein SDK ist eine Runtime, die zusätzlich Entwicklungswerkzeuge, Header-Dateien, Compiler und Debugger enthält. Jede Anwendung wird gegen ein SDK gebaut, das typischerweise mit einer Runtime gekoppelt ist, welche zur Laufzeit von der Anwendung genutzt wird.

Die Sicherheit wird durch die **Sandbox-Technologie** von Flatpak erhöht. Jede Anwendung läuft in einer isolierten Umgebung und hat standardmäßig nur Zugriff auf sich selbst und ihre Runtime. Zugriffe auf Benutzerdateien, Netzwerk, Grafiksockets und Geräte müssen explizit gewährt werden.[12]

Flatpak baut auf bestehenden **Technologien** auf, darunter bubblewrap, systemd, D-Bus, das OCI-Format der Open Container Initiative, OSTree und AppStream-Metadaten. Diese Komponenten tragen dazu bei, dass Flatpak-Anwendungen nahtlos in Software-Center-Anwendungen integriert werden können.[13]

Die Verwaltung von Flatpak-Anwendungen erfolgt über das Kommandozeilenwerkzeug **flatpak**. Mit diesem können Runtimes und Anwendungen installiert, entfernt und aktualisiert werden. Es bietet zudem Funktionen zum Anzeigen der installierten Pakete sowie zum Bauen und Verteilen von Anwendungs-Bundles. Standardmäßig werden die meisten flatpak-Befehle systemweit ausgeführt; mit der Option –user können sie jedoch auch nur für den aktuellen Benutzer angewendet werden.[11]

Das Konzept von Flatpak basiert auf der Idee eines flexiblen und sicheren Systems zur Verteilung und Ausführung von Anwendungen unter Linux. Dabei ist es unerheblich, welche Distribution zum Einsatz kommt. Die Isolation der Anwendungen sowie die Möglichkeit, spezifische Runtimes und Abhängigkeiten zu nutzen, zielen darauf ab, sowohl die Kompatibilität als auch die Sicherheit zu erhöhen.

3.3.1 Verpakungsprozzes

Flatpak-Bauumgebung

Um die zu erstellende Flatpak-Anwendung auszuführen oder zu testen, wird **flatpak** benötigt. Es kümmert sich um die Installation und Ausführung des Flatpak-Pakets und verwaltet außerdem Laufzeiten und Berechtigungen.

Die Erstellung einer Flatpak-Anwendung ist komplexer als das einfache Verpacken von Code. Es müssen Abhängigkeiten ermittelt, die Umgebung konfiguriert und die Ressourcen korrekt exportiert werden. **flatpak-builder** automatisiert diesen Prozess, indem es den Build durchführt, die notwendigen Konfigurationen vornimmt und die fertige Anwendung packt.

```
$ sudo apt install flatpak\
$ sudo apt install flatpak-builder
```

Flatpak-Anwendungen verlassen sich auf gemeinsame Laufzeiten, um eine standardisierte Basisumgebung bereitzustellen.

```
$ flatpak remote-add --if-not-exists flathub
   https://flathub.org/repo/flathub.flatpakrepo\
$ flatpak install flathub org.gnome.Platform//47\
$ flatpak install flathub org.gnome.Sdk//47
```

Erstellen einer YAML

Die Konstruktion jedes Flatpaks hängt von einer Manifestdatei ab, die grundlegende Informationen über die Art der Anwendung und die für ihre Erstellung erforderlichen Anweisungen enthält.[14]

Mit folgendem Befehle wird ein flatpak manifest in seinem bevorzugte Verzeichnis erstellt.

```
$ cd path/to/prefDir
$ mkdir dia-flatpak
$ cd dia-flatpak
$ touch dia.flatpak.yml
```

Nachdem das Manifest erstellt ist, muss es konfiguriert werden, was im Folgenden beschrieben wird.

```
app-id: org.gnome.Dia
runtime: org.gnome.Platform
runtime-version: "47"
sdk: org.gnome.Sdk
command: dia
```

- **app-id** wird verwendet, um die App im Flatpak-System zu identifizieren und muss innerhalb des Flatpak-Ökosystems eindeutig sein.
- **runtime** enthält Komponenten, die häufig von GNOME-Anwendungen verwendet werden. Die Verwendung einer Laufzeitumgebung reduziert die Größe des Flatpak-Pakets und stellt die Konsistenz zwischen verschiedenen Installationen sicher.
- **runtime-version** gibt die Version der Laufzeitumgebung an.
- **sdk** enthält Entwicklungswerkzeuge, Header und Bibliotheken, die für die Erstellung der Anwendung benötigt werden.
- **command** gibt den ausführbaren Befehl zum Starten der Anwendung an. Nach der Installation führt der Aufruf von *flatpak run org.gnome.Dia* dia innerhalb der Flatpak-Sandbox aus.

```
finish-args:
  - --socket=x11
  - --socket=wayland
  - --filesystem=xdg-data/applications
  - --filesystem=host
```

- **finish-args** definiert die Berechtigungen und Ressourcen, die die Anwendung außerhalb ihrer Sandbox benötigt:
- **–socket=x11:** Ermöglicht der Anwendung den Zugriff auf den X11-Anzeigeserver, so dass sie eine grafische Oberfläche anzeigen kann.
- **–socket=wayland:** Ermöglicht es der Anwendung, den Wayland-Anzeigeserver für moderne grafische Oberflächen zu verwenden.
- **–filesystem=xdg-data/applications:** Ermöglicht den Zugriff auf Anwendungsdatenverzeichnisse und damit die Interaktion mit anderen Anwendungsdaten.
- **–filesystem=host:** Bietet einen breiteren Zugriff auf das Dateisystem, nützlich für den Zugriff auf gemeinsam genutzte Dateie

```
modules:
  - name: xpm-pixbuf
    buildsystem: meson
    config-opts: []
    sources:
      - type: git
        url: https://gitlab.gnome.org/ZanderBrown/xpm-pixbuf.git
        branch: main

  - name: dia
    buildsystem: meson
    config-opts: []
    sources:
      - type: git
        url: https://gitlab.gnome.org/GNOME/dia.git
        branch: master
```

Xpm-pixbuf

- **name: xpm-pixbuf** ist eine von Dia benötigte Abhängigkeit, die nicht in Runtime enthalten ist, um XPM-Bilder zu verarbeiten.
- **buildsystem: meson** Gibt an, dass dieses Modul mit dem Meson-Buildsystem erstellt wird.
- **sources:** Verweist auf das Git-Repository, in dem xpm-pixbuf geklont und kompiliert werden kann.

Dia (Hauptanwendung)

- **name: dia** Dies ist das Hauptmodul, das die Dia-Anwendung repräsentiert.
- **buildsystem: meson** Dia wird mit dem Meson-Buildsystem erstellt.
- **sources:** Verweist auf das Git-Repository der Dia-Anwendung, aus dem Flatpak den Quellcode holt.

3.3.2 Vollständige dia.flatpak.yaml Datei

```
app-id: org.gnome.Dia
runtime: org.gnome.Platform
runtime-version: "47"
sdk: org.gnome.Sdk
command: dia
finish-args:
  - --socket=x11
  - --filesystem=xdg-data/applications
  - --filesystem=host
  - --share=network
  - --socket=wayland
modules:
  - name: xpm-pixbuf
    buildsystem: meson
    config-opts: []
    sources:
      - type: git
        url: https://gitlab.gnome.org/ZanderBrown/xpm-pixbuf.git
        branch: main

  - name: dia
    buildsystem: meson
    config-opts: []
    sources:
      - type: git
        url: https://gitlab.gnome.org/GNOME/dia.git
        branch: master
```

Bau des Dia-Flatpak

Nachdem die Yaml-Datei fertiggestellt war, war er bereit, den Dia-Flatpak zu bauen. Dazu gibt es zwei Installationswege. Der erste Weg besteht darin, den Erstellungsprozess zu initialisieren, der dann automatisch von Snapcraft verwaltet wird. Dieser Prozess wird mit dem folgenden Befehl im Stammverzeichnis des zu erstellenden Snap gestartet, in dem sich auch die Yaml-Datei befindet:

Es gibt zwei Möglichkeiten, die App zu installieren:

1. Der erste Weg ist die lokale Installation mit der Option –install und dem folgenden Befehl:

```
$ flatpak-builder (--force-clean) --user --install
   build-dir dia.flatpak.yml
```

2. Die zweite Möglichkeit besteht darin, in ein Repository zu exportieren, ein eigenständiges .flatpak-Bundle aus dem lokalen Repository zu erstellen und es dann lokal aus der erstellten .flatpak-Datei zu installieren. Die .flatpak-Datei, die mit dem zweiten Befehl erstellt wird, enthält alle notwendigen Dateien, Abhängigkeiten und Metadaten, so dass es ein eigenständiges Paket ist, das auf jedem Flatpak-unterstützten System installiert werden kann, ohne das ursprüngliche Repository zu benötigen. Es kann als eine einzige Datei verteilt werden, so dass es leicht auf anderen Systemen weitergegeben oder getestet werden kann.

```
$ flatpak-builder --repo=repo --force-clean build-dir
   dia.flatpak.yml
$ flatpak build-bundle repo dia.flatpak org.gnome.Dia
$ flatpak install --user --dangerous dia.flatpak
```

Die erste Methode wird hauptsächlich zum lokalen Testen der Anwendung verwendet, während die zweite Methode ein Repository erstellt, das bereit ist, in das Haupt-Repository von Flathub übertragen zu werden.

Nach diesen Schritten war Dia schließlich als Snap verpackt und funktionierte wie jede andere Anwendung.

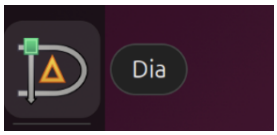

ABBILDUNG 3.6: Dia(Flatpak)

3.3.3 Verteilung an anderen Linux Distributionen

Der erste Schritt besteht darin, die gepackte Standalone-Datei dia.flatpak mit einem USB-Stick oder einer anderen Methode auf einen anderen Ubuntu-PC zu übertragen.

Zur Installation des Pakets wird das Programm flatpak benötigt, das mit folgendem Befehl installiert wird:

LISTING 3.2: dnf für Fedora and pacman für Manjaro Linux

```
$ sudo apt install flatpak
```

Flathub Repository instllieren damit Runtime und SDK zu intstallieren

```
$ flatpak remote-add --if-not-exists --user flathub\
   https://dl.flathub.org/repo/flathub.flatpakrepo
```

```
$ flatpak install flathub org.gnome.Platform//47\
$ flatpak install flathub org.gnome.Sdk//47
```

Nach diese Schritten kann man jetzt das dia.flatpak Detei installieren.

```
$ flatpak install --user --dangerous dia.flatpak
```

3.4 Snap

Das Snap-Packaging-System packt alle für eine Anwendung erforderlichen Dateien in eine einzige Snap-Verteilungsdatei.

Der snap-Daemon **snapd** verwaltet die auf dem System installierten Snap-Pakete und läuft im Hintergrund. Mit dem Kommandozeilentool snap wird die snap-Datenbank abgefragt, um die installierten Snap-Pakete anzuzeigen, sowie Snap-Pakete zu installieren, zu aktualisieren und zu entfernen.[15]

Das Tool **Snapcraft** erleichtert die Erstellung, Verpackung und Veröffentlichung von Snaps. Es ist mit Ubuntu, zahlreichen anderen Linux-Distributionen und macOS kompatibel und kann über die Kommandozeile ausgeführt werden.

Die Verwendung von Snapcraft ermöglicht es Entwicklern, plattformspezifische Plugins und Erweiterungen einzusetzen und so den Build-Prozess zu vereinfachen und zu rationalisieren. Anschließend können Snaps lokal getestet und geteilt werden, bevor sie im globalen Snap Store innerhalb von Channels, Tracks und Branches veröffentlicht werden, wodurch eine präzise Kontrolle über die Releases möglich ist.

Snapcraft führt all diese Funktionen innerhalb eines selbstgestarteten LXD- oder Multipass-Containers oder nativ in der eigenen Umgebung des Benutzers aus. Darüber hinaus enthält es Linting-, Debug- und Staging-Funktionen, die bei der Behebung von Problemen helfen. Darüber hinaus kann Snapcraft in bestehende CI-Systeme integriert werden und ermöglicht die Remote-Erstellung von Snaps von GitHub oder unserer Launchpad-Build-Farm.

Die Vielseitigkeit von Snapcraft erstreckt sich auf eine Vielzahl von Szenarien, darunter Desktop-Anwendungen, Server, Cloud-Bereitstellungen, eingebettete Geräte und IoT. Die Fähigkeiten von Snapcraft sind nicht durch die Anzahl der Benutzer begrenzt, sondern reichen von einer einzelnen Person bis zu Millionen.

3.4.1 Verpackungsprozess

Snapcraft-Bauumgebung

Um mit dem Packen zu beginnen, war es zunächst erforderlich, die Anweisungen aus der Snapcraft-Dokumentation zu befolgen. Zunächst musste das Snap installiert werden, da Snapcraft selbst ein Snap-Paket ist und aus der

Snap-store installiert werden kann.

```
$ sudo apt install snap
```

Nach erfolgreicher Installation von 'snap' konnte die Anwendung ßnapcraftünter Verwendung des folgenden Kommandos installiert werden:

```
$ sudo snap install snapcraft --classic
```

Die Erstellung von sogenannten 'Snaps' erfolgt standardmäßig in einer LXD-Container-Umgebung durch das Tool 'Snapcraft'. Dies gewährleistet, dass ein Snap-Build vom Arbeitssystem isoliert wird und sichergestellt ist, dass alle Abhängigkeiten, die ein Snap benötigt, ausschließlich durch den Build-Prozess bereitgestellt werden.

Zunächst war es erforderlich, die LXD-Build-Umgebung zu installieren und die aktuelle Benutzersitzung zur LXD-Gruppe hinzuzufügen, um auf die Ressourcen zugreifen zu können. Dies wurde wie folgt durchgeführt:

```
# lxd installieren
$ sudo snap install lxd
# Benutzersitzung zur LXD-Gruppe hinzufuegen
$ sudo usermod -a -G lxd $USER
```

Der letzte Schritt beim Einrichten von Snapcraft und der Build-Umgebung bestand darin, die lxd-Umgebung zu initialisieren.

```
$ lxd init --minimal
```

Erstellen einer YAML

Snapcraft nutzt eine snapcraft.yaml-Datei, um festzulegen, wie ein Snap-Paket erstellt wird und welche Inhalte es hat. Diese YAML-Datei ist quasi die Bauplan für die Erstellung des Snap-Pakets. Sie enthält wichtige Abschnitte wie:

Name, Version, Zusammenfassung und Beschreibung: Grundlegende Metadaten über den Snap, wie Name, Version und eine kurze Beschreibung.

base: Gibt die Laufzeitumgebung an, z. B. core22 oder core24, die die zugrunde liegenden Bibliotheken und die Kompatibilität bestimmt.

```
name: dia
base: core24
version: "0.1"
summary: Dia diagram editor
description: >
Dia is a program to draw structured diagrams, such as \
flowcharts,network diagrams, and more. This snap packages\
the Dia app along with allrequired dependencies for a self-\
```

```
contained deployment.
```

Confinement: bestimmt Der Grad des Zugriffs einer Anwendung auf Systemressourcen wie Dateien, das Netzwerk, Peripheriegeräte und Dienste. Es gibt mehrere Stufen der Beschränkung, die in Betracht gezogen werden können.

Die Mehrzahl der Snaps wird **'strict'** genutzt. Es wird empfohlen, dass streng eingeschränkte Snaps in vollständiger Isolation bis zu einer minimalen Zugriffsstufe laufen, die als immer sicher angesehen wird. Streng eingeschränkte Snaps können daher nicht auf Dateien, Netzwerke, Prozesse oder andere Systemressourcen zugreifen, ohne einen spezifischen Zugriff über eine Schnittstelle (Plugs) anzufordern.

gnome-3-38-2004: ermöglicht die Verwendung von Gsettings-Schemata, die von GTK-Anwendungen wie Dia benötigt werden.
x11: Der x11-Plug ermöglicht der Anwendung die Anzeige ihrer grafischen Oberfläche, indem er ihr Zugriff auf den X11-Anzeigeserver gewährt.
home: Dia muss die Möglichkeit haben, Dateien in das Home-Verzeichnis zu öffnen und zu speichern.
opengl: Auch wenn es sich bei Dia nicht um eine grafikorientierte Anwendung handelt, kann die Verwendung von OpenGL die Rendering-Leistung verbessern und flüssigere Interaktionen ermöglichen.

```
confinement: strict
.
.
.
apps:
  dia:
    plugs:
      - gnome-3-38-2004
      - x11
      - home
      - opengl
```

Layouts: verändern die Ausführungsumgebung eines streng begrenzten Snaps.

/usr/local/share/dia: Dies ist der Pfad, in dem Dia seine Ressourcen erwartet, wie Konfigurationsdateien, Plugins, Symbole und andere notwendige Datendateien, wenn es auf einem Standard-System läuft.

bind: $SNAP/usr/local/share/dia: Dieser Befehl erstellt eine Bindung, die im Wesentlichen Anfragen für **/usr/local/share/dia** auf das entsprechende Verzeichnis innerhalb des Snap-Pakets umleitet, **$SNAP/usr/local/share/dia**. **$SNAP** ist eine Variable, die auf das Stammverzeichnis des installierten Snap-Pakets verweist.

```
layout:
```

```
    /usr/local/share/dia:
    bind: $SNAP/usr/local/share/dia
```

Parts enthalten Dateien und Abhängigkeiten, die in den endgültigen Snap integriert werden und zur Laufzeit zugänglich sind. Ein Teil sollte erstellt werden, wenn die Komponente, die sie bereitstellt, von der Anwendung benötigt wird, während sie läuft oder baut.

Ninja Part

```
  parts:
    ninja:
      plugin: nil
      source: https://github.com/ninja-build/ninja.git
      source-depth: 1
      source-tag: v1.12.1
      build-packages:
        - python3
        - python3-setuptools
        - python3-pip
```

Der Ninja-Teil hat die Aufgabe, das Ninja-Build-System zu bauen und zu installieren. Ninja ist ein schnelles und effizientes Build-System, das häufig als Backend für andere Build-Tools wie Meson verwendet wird.

plugin: nil bedeutet, dass kein spezifisches Snapcraft-Plugin verwendet wird, so dass ein direkter Build aus dem Quellcode möglich ist. **Source** verweist auf das GitHub-Repository von Ninja mit der Version v1.12.1. **build-packages** enthalten Python-Tools (python3, python3-setuptools, python3-pip), die für die Einrichtung der Build-Umgebung und der Abhängigkeiten von Ninja erforderlich sind.

Meson-Deps Part

```
    meson-deps:
      after: [ninja]
      plugin: nil
      override-build: |
        # Install Meson from the source
        git clone https://github.com/mesonbuild/meson.git
        cd meson
        git checkout 1.4.1
        python3 setup.py install --prefix=$SNAPCRAFT_PART_INSTALL
      build-packages:
        - python3
        - python3-setuptools
        - python3-pip
```

Dieser Teil installiert das Meson-Build-System, ein High-Level-Tool, das den Prozess der Softwareerstellung vereinfacht. Meson erzeugt Ninja-Dateien,

die Ninja dann zum Kompilieren des Projekts verwendet.

after: [ninja] stellt sicher, dass meson-deps nach ninja gebaut wird, da Meson von Ninja abhängt.

override-build klont den Quellcode von Meson, checkt die Version 1.4.1 aus und installiert sie in der Snap-Umgebung.

build-packages (Python-Tools: python3, python3-setuptools und python3-pip) werden benötigt, um die Abhängigkeiten und das Setup von Meson zu verwalten.

Glib Part

```
glib:
  plugin: meson
  source: https://gitlab.gnome.org/GNOME/glib.git
  source-depth: 1
  source-type: git
  source-tag: 2.80.0
  meson-parameters:
    - -Dtests=false
    - --wrap-mode=nodownload
```

GLib ist eine grundlegende Bibliothek, die wesentliche Datenstrukturen, Dienstprogramme und Funktionen bereitstellt, die im GNOME-Ökosystem und anderen GTK-basierten Anwendungen weit verbreitet sind.

Das **plugin** meson wird verwendet, um GLib zu bauen, da es mit Meson konfiguriert ist. **Source** verweist auf das GLib-Repository und gibt die Version 2.80.0 an. Um den Build-Prozess zu beschleunigen, deaktiviert Meson-parameters die Tests **(-Dtests=false)** und setzt **–wrap-mode=nodownload**, um das automatische Herunterladen von Abhängigkeiten zu verhindern.

Xpm-pixbuf Part

```
xpm-pixbuf:
  after: [glib]
  plugin: meson
  source: https://gitlab.gnome.org/ZanderBrown/xpm-pixbuf.git
  source-type: git
  build-packages:
    - libgdk-pixbuf2.0-dev
    - libglib2.0-dev
    - git
    - meson
    - ninja-build
    - pkg-config

  stage-packages:
    - libxpm4 # Ensure runtime dependency for xpm
    - libglib2.0-0
    - libc6
    - libgdk-pixbuf2.0-0
```

xpm-pixbuf ist ein Bildlader, der Unterstützung für das XPM-Bildformat in GTK-Anwendungen bietet. XPM ist ein einfaches Bildformat, das häufig für Icons in Linux-Anwendungen verwendet wird.

after: [glib] stellt sicher, dass xpm-pixbuf nach glib gebaut wird, da es davon abhängt. **plugin: meson** verwendet das Meson-Plugin für den Build, da xpm-pixbuf Meson-konfiguriert ist. **build-packages** enthält Entwicklungsbibliotheken und Build-Tools, die für die Kompilierung erforderlich sind. **stage-packages** enthält Laufzeitbibliotheken (libxpm4, libglib2.0-0, libc6, libgdk-pixbuf2.0-0), um sicherzustellen, dass xpm-pixbuf in der Snap-Umgebung korrekt funktioniert.

Dia Part

```
dia:
  after: [meson-deps, glib, xpm-pixbuf]
  source: https://gitlab.gnome.org/GNOME/dia.git
  source-depth: 1
  plugin: meson
  build-packages:
    - cmake
    - meson
    .
    .
    .
  stage-packages:
    - libxml2
    - zlib1g
    .
    .
    .
  override-build: |
    craftctl default
    /root/parts/dia/install/usr/lib/aarch64-linux-gnu/gdk-pixbuf-2.0/
    gdk-pixbuf-query-loaders > $SNAPCRAFT_PART_INSTALL/usr/lib/

    aarch64-linux-gnu/gdk-pixbuf-2.0/2.10.0/loaders.cache

    glib-compile-schemas
       $SNAPCRAFT_PART_INSTALL/usr/share/glib-2.0/schemas

    $SNAPCRAFT_PART_INSTALL/usr/bin/fc-cache -v -f
       $SNAPCRAFT_PART_INSTALL/usr/share/fonts

    $SNAPCRAFT_PART_INSTALL/usr/bin/update-mime-database
       $SNAPCRAFT_PART_INSTALL/usr/share/mime
```

Dia **part** stützt sich bei der Erstellung auf andere Teile (meson-deps, glib, xpm-pixbuf) und verwendet sowohl build-packages als auch stage-packages, um notwendige Bibliotheken und Werkzeuge einzubinden. Der Abschnitt override-build stellt sicher, dass die Laufzeitressourcen von Dia, wie z. B. Bildlader, Schemata, Schriftarten und MIME-Typen, korrekt konfiguriert sind, so dass der Snap in sich geschlossen und in einer begrenzten Umgebung funktionsfähig ist.

Vollständige snapcraft.yaml Datei

```
name: dia
base: core24
version: "0.1"
summary: Dia diagram editor
description: >
  Dia is a program to draw structured diagrams, such as
     flowcharts,
  network diagrams, and more. This snap packages the Dia app
     along with all
  required dependencies for a self-contained deployment.
grade: stable
confinement: strict

layout:
  /usr/local/share/dia:
    bind: $SNAP/usr/local/share/dia

parts:
  ninja:
    plugin: nil
    source: https://github.com/ninja-build/ninja.git
    source-depth: 1
    source-tag: v1.12.1
    build-packages:
      - python3
      - python3-setuptools
      - python3-pip

  meson-deps:
    after: [ninja]
    plugin: nil
    override-build: |
      # Install Meson from the source
      git clone https://github.com/mesonbuild/meson.git
      cd meson
      git checkout 1.4.1
      python3 setup.py install --prefix=$SNAPCRAFT_PART_INSTALL
    build-packages:
      - python3
      - python3-setuptools
      - python3-pip

  glib:
    plugin: meson
    source: https://gitlab.gnome.org/GNOME/glib.git
    source-depth: 1
    source-type: git
```

```
    source-tag: 2.80.0
    meson-parameters:
      - -Dtests=false
      - --wrap-mode=nodownload

  xpm-pixbuf:
    after: [glib]
    plugin: meson
    source: https://gitlab.gnome.org/ZanderBrown/xpm-pixbuf.git
    source-type: git
    build-packages:
      - libgdk-pixbuf2.0-dev
      - libglib2.0-dev
      - git
      - meson
      - ninja-build
      - pkg-config

    stage-packages:
      - libxpm4 # Ensure runtime dependency for xpm
      - libglib2.0-0
      - libc6
      - libgdk-pixbuf2.0-0

  dia:
    after: [meson-deps, glib, xpm-pixbuf]
    source: https://gitlab.gnome.org/GNOME/dia.git
    source-depth: 1
    plugin: meson
    build-packages:
      - cmake
      - meson
      - ninja-build
      - build-essential
      - python3
      - python-gi-dev
      - xsltproc
      - libxslt1-dev
      - libxml2-dev
      - libgtk-3-dev
      - libgraphene-1.0-dev
      - zlib1g-dev
      - libxslt1-dev
      - gettext
      - desktop-file-utils
      - appstream
      - python3-dev
      - libgtk-4-dev
      - dblatex
```

```
      - patchelf
    stage-packages:
      - libxml2
      - zlib1g
      - libgtk-3-0
      - libxslt1.1
      - python3
      - python3-gi
      - libgraphene-1.0-0
      - libc6
      - libpython3-dev
      - libgtk-4-1
      - libgdk-pixbuf2.0-0
      - yaru-theme-gtk # Updated theme package
      - adwaita-icon-theme
      - librsvg2-common
      - libcanberra-gtk3-module
      - libcanberra-gtk-module
      - libcanberra0

    override-build: |
      craftctl default
      /root/parts/dia/install/usr/lib/aarch64-linux-gnu/gdk-pixbuf-2.0/\
      gdk-pixbuf-query-loaders >
          $SNAPCRAFT_PART_INSTALL/usr/lib/aarch64-\
      linux-gnu/gdk-pixbuf-2.0/2.10.0/loaders.cache
      glib-compile-schemas
          $SNAPCRAFT_PART_INSTALL/usr/share/glib-2.0/schemas
      $SNAPCRAFT_PART_INSTALL/usr/bin/fc-cache -v -f
          $SNAPCRAFT_PART_INSTALL/\
      usr/share/fonts
      $SNAPCRAFT_PART_INSTALL/usr/bin/update-mime-database
          $SNAPCRAFT_PART_INSTALL\
      /usr/share/mime

apps:
  dia:
    command: usr/local/bin/dia
    environment:
      XDG_DATA_HOME: "$SNAP_USER_DATA/.local/share"
      GDK_PIXBUF_MODULE_FILE:
          "$SNAP/usr/lib/aarch64-linux-gnu/gdk-pixbuf-2.0\
      /2.10.0/loaders.cache"
      GDK_PIXBUF_MODULEDIR:
          "$SNAP/usr/lib/aarch64-linux-gnu/gdk-pixbuf-2.0/\
      2.10.0/loaders"
      DIA_UI_PATH: "$SNAP/usr/local/share/dia/ui"
      PYTHONPATH: $SNAP/usr/local/share/dia/
```

```
      PYDIA_STARTUP_FILE:
         "$SNAP/usr/local/share/dia/python-startup.py"
      XDG_DATA_DIRS: "$SNAP/usr/share:/usr/share"
      SNAP_DESKTOP_RUNTIME: $SNAP/gnome-platform
      DIA_LIB_PATH: "$SNAP/usr/local/lib/"
      LD_LIBRARY_PATH:
         "$SNAP/usr/local/lib:$SNAP/usr/local/lib/aarch64-linux\
      -gnu:$LD_LIBRARY_PATH"

    plugs:
      - gnome-3-38-2004
      - x11
      - home
      - opengl

plugs:
  gnome-3-38-2004:
    interface: content
    target: $SNAP/gnome-platform
    default-provider: gnome-3-38-2004
```

3.4.2 Evaluierung der Möglichkeiten zur Verkleinerung der Verpackung von Snap

Nach der Erstellung des Snap-Pakets konnte ein signifikanter Größenunterschied im Vergleich zu den beiden anderen Verpackungsmethoden festgestellt werden. Die erstellte Snap-Paketgröße betrug 72,3 MiB.

In der Folge wurde eine Untersuchung durchgeführt, um die Gründe für den Größenunterschied zwischen Snap und den anderen Verpackungsmethoden zu ermitteln. Dabei wurden insbesondere die Faktoren untersucht, die dazu beitragen, einschließlich der Abhängigkeitsbündelung, Laufzeitbibliotheken und des für jede Methode spezifischen Verpackungsaufwands.

In der Dokumentation von Snapcraft gibt es einen ganzen Abschnitt, in dem beschrieben wird, wie ein Snap in der Größe reduziert werden kann.

Verwendung von gnome-erweiterung

Beginnend mit der Verwendung der Gnome-Erweiterung[16]. Snapcraft-Erweiterungen erleichtern die Integration einer Reihe gemeinsamer Anforderungen in ein Snap für Snap-Entwickler. Die Verwendung der Gnome-Erweiterung stellt sicher, dass die GTK3- und GNOME-Bibliotheken (glib, Cursor-Themen) für alle Teile zur Erstellungs- und Laufzeit verfügbar sind.

Um den Snap angewiesen werden zu können, Gnome zu verwenden, um die erforderlichen Bibliotheken abzurufen, ist es erforderlich, die manuellen Abhängigkeiten (ninja, meson-deps und glib) aus der Datei ßnapcraft.yamlßu entfernen. Anstelle dessen muss lediglich die folgende Zeile in den Apps-Teil der Datei snapcraft.yaml hinzugefügt werden.

```
  apps:
dia:
  command: usr/bin/dia
  extensions: [gnome]
  :
  :
```

Cleanup part

Im zweiten Schritt ist ein Cleanup part zu erstellen, welcher das Löschen der nicht verwendeten Bibliotheken ermöglicht.

```
cleanup:
  after: [xpm-pixbuf, dia]
  plugin: nil
  override-prime: |
    set -eux
    for lib in \
      usr/lib/aarch64-linux-gnu/libBrokenLocale.so.1 \
```

```
      usr/lib/aarch64-linux-gnu/libanl.so.1 \
      usr/lib/aarch64-linux-gnu/libc_malloc_debug.so.0 \
      usr/lib/aarch64-linux-gnu/libdl.so.2 \
      usr/lib/aarch64-linux-gnu/libexslt.so.0.8.21 \
      usr/lib/aarch64-linux-gnu/libgdk_pixbuf_xlib-2.0.so.\
      0.4000.2 \
      usr/lib/aarch64-linux-gnu/libicuio.so.74.2 \
      usr/lib/aarch64-linux-gnu/libicutest.so.74.2 \
      usr/lib/aarch64-linux-gnu/libmemusage.so \
      usr/lib/aarch64-linux-gnu/libmvec.so.1 \
      usr/lib/aarch64-linux-gnu/libnsl.so.1 \
      usr/lib/aarch64-linux-gnu/libnss_compat.so.2 \
      usr/lib/aarch64-linux-gnu/libnss_dns.so.2 \
      usr/lib/aarch64-linux-gnu/libnss_files.so.2 \
      usr/lib/aarch64-linux-gnu/libnss_hesiod.so.2 \
      usr/lib/aarch64-linux-gnu/libpcprofile.so \
      usr/lib/aarch64-linux-gnu/libproxy.so.0.5.4 \
      usr/lib/aarch64-linux-gnu/libpthread.so.0 \
      usr/lib/aarch64-linux-gnu/librt.so.1 \
      usr/lib/aarch64-linux-gnu/libthread_db.so.1 \
      usr/lib/aarch64-linux-gnu/libicutu.so.74.2\
      usr/lib/aarch64-linux-gnu/libutil.so.1; do
    if [ -e $SNAPCRAFT_PRIME/$lib ]; then
       rm $SNAPCRAFT_PRIME/$lib
    fi
    done
```

Anpassungen der Layoutdefinition

Im Anschluss an die Installation der GNOME-Erweiterung waren einige Modifikationen in der Layoutdefinition erforderlich.

Die GNOME-Erweiterung generiert eine vordefinierte Laufzeitumgebung, wobei die Applikation unter Umständen nicht immer in der Lage ist, ihre Ressourcen automatisch zu identifizieren und zuzuordnen.

Um den spezifischen Laufzeiterwartungen von dia gerecht zu werden, sind zusätzliche Layout-Bindungen erforderlich.

```
layout:
  /usr/local/share/dia:
  bind: $SNAP/usr/share/dia

  /snap/dia/x1/usr/local/share/dia/data/ui:
  bind: $SNAP/usr/share/dia/ui

  /snap/dia/x1/usr/share/dia/python/python-startup.py:
  bind-file: $SNAP/usr/share/dia/python-startup.py
```

Nach Abschluss der drei wesentlichen Aktualisierungen konnte die als Snap verpackte Dia mit einer Größe von 57,7 Mib erstellt werden.

Die geänderte Version von snapcraft.yaml

```
name: dia
base: core24
version: "0.1"
summary: Dia diagram editor
description: >
  Dia is a program to draw structured diagrams, such as
     flowcharts,
  network diagrams, and more. This snap packages the Dia app
     along with all
  required dependencies for a self-contained deployment.
grade: stable
confinement: strict

layout:
  /usr/local/share/dia:
    bind: $SNAP/usr/share/dia

  /snap/dia/x1/usr/local/share/dia/data/ui:
    bind: $SNAP/usr/share/dia/ui

  /snap/dia/x1/usr/share/dia/python/python-startup.py:
    bind-file: $SNAP/usr/share/dia/python-startup.py

parts:
name: dia
base: core24
version: "0.1"
summary: Dia diagram editor
description: >
  Dia is a program to draw structured diagrams, such as
     flowcharts,
  network diagrams, and more. This snap packages the Dia app
     along with all
  required dependencies for a self-contained deployment.
grade: stable
confinement: strict

layout:
  /usr/local/share/dia:
    bind: $SNAP/usr/share/dia

  /snap/dia/x1/usr/local/share/dia/data/ui:
    bind: $SNAP/usr/share/dia/ui
```

```
  /snap/dia/x1/usr/share/dia/python/python-startup.py:
    bind-file: $SNAP/usr/share/dia/python-startup.py

parts:
  xpm-pixbuf:
    plugin: meson
    source: https://gitlab.gnome.org/ZanderBrown/xpm-pixbuf.git
    source-type: git
    meson-parameters:
      - --prefix=/usr
      - --buildtype=release
      - '-Dc_args=-Wno-error=format-nonliteral
         -Wno-error=format=2'
      - '-Dc_link_args=-Wno-error=format-nonliteral
         -Wno-error=format=2'
    build-packages:
      - libglib2.0-dev
      - libgdk-pixbuf2.0-dev
      - pkg-config
      - ninja-build
    stage-packages:
      - libglib2.0-0
      - libgdk-pixbuf2.0-0
    override-build: |
      set -eux
      cd $SNAPCRAFT_PART_SRC
      # Remove -Werror=format-nonliteral from meson.build
      sed -i 's/-Werror=format-nonliteral//g;
         s/-Werror=format=2//g' meson.build
      meson setup --wipe build --prefix=/usr \
        '-Dc_args=-Wno-error=format-nonliteral
           -Wno-error=format=2' \
        '-Dc_link_args=-Wno-error=format-nonliteral
           -Wno-error=format=2'
      ninja -C build
      DESTDIR=$SNAPCRAFT_PART_INSTALL ninja -C build install

  dia:
    after: [xpm-pixbuf]
    source: https://gitlab.gnome.org/GNOME/dia.git
    source-depth: 1
    plugin: meson
    meson-parameters:
      - --prefix=/usr
      - --buildtype=release
      - '-Dc_args=-Wno-error=format-nonliteral
         -Wno-error=format=2'
      - '-Dc_link_args=-Wno-error=format-nonliteral
         -Wno-error=format=2'
```

```
build-packages:
  - cmake
  - meson
  - ninja-build
  - build-essential
  - python3
  - python-gi-dev
  - xsltproc
  - libxml2-dev
  - libgtk-3-dev
  - libgraphene-1.0-dev
  - zlib1g-dev
  - libxslt1-dev
  - gettext
  - desktop-file-utils
  - appstream
  - python3-dev
  - libgtk-4-dev
  - dblatex
  - patchelf
  - adwaita-icon-theme
stage-packages:
  - libxml2
  - zlib1g
  - libgtk-3-0
  - libxslt1.1
  - python3
  - python3-gi
  - libgraphene-1.0-0
  - libc6
  - libgtk-4-1
  - libgdk-pixbuf2.0-0
  - yaru-theme-gtk # Updated theme package
  - adwaita-icon-theme
  - libcanberra-gtk3-module
  - librsvg2-common
  - libcanberra-gtk-module
  - libproxy1v5
  - libpixman-1-0
  - libproxy1v5
build-attributes:
  - enable-patchelf
override-build: |
  craftctl default
  $SNAPCRAFT_PART_INSTALL/usr/bin/fc-cache -v -f
     $SNAPCRAFT_PART_INSTALL/usr/share/fonts
  $SNAPCRAFT_PART_INSTALL/usr/bin/update-mime-database
     $SNAPCRAFT_PART_INSTALL/usr/share/mime
```

```
      glib-compile-schemas
         $SNAPCRAFT_PART_INSTALL/usr/share/glib-2.0/schemas
      mv $SNAPCRAFT_PART_INSTALL/usr/share/dia/python-startup.py
         $SNAPCRAFT_PART_INSTALL/usr/share/dia/python/

  cleanup:
    after: [xpm-pixbuf, dia]
    plugin: nil
    override-prime: |
      set -eux
      for lib in \
        usr/lib/aarch64-linux-gnu/libBrokenLocale.so.1 \
        usr/lib/aarch64-linux-gnu/libanl.so.1 \
        usr/lib/aarch64-linux-gnu/libc_malloc_debug.so.0 \
        usr/lib/aarch64-linux-gnu/libdl.so.2 \
        usr/lib/aarch64-linux-gnu/libexslt.so.0.8.21 \
        usr/lib/aarch64-linux-gnu/libgdk_pixbuf_xlib-2.0.so.0.4000.2
           \
        usr/lib/aarch64-linux-gnu/libicuio.so.74.2 \
        usr/lib/aarch64-linux-gnu/libicutest.so.74.2 \
        usr/lib/aarch64-linux-gnu/libmemusage.so \
        usr/lib/aarch64-linux-gnu/libmvec.so.1 \
        usr/lib/aarch64-linux-gnu/libnsl.so.1 \
        usr/lib/aarch64-linux-gnu/libnss_compat.so.2 \
        usr/lib/aarch64-linux-gnu/libnss_dns.so.2 \
        usr/lib/aarch64-linux-gnu/libnss_files.so.2 \
        usr/lib/aarch64-linux-gnu/libnss_hesiod.so.2 \
        usr/lib/aarch64-linux-gnu/libpcprofile.so \
        usr/lib/aarch64-linux-gnu/libproxy.so.0.5.4 \
        usr/lib/aarch64-linux-gnu/libpthread.so.0 \
        usr/lib/aarch64-linux-gnu/librt.so.1 \
        usr/lib/aarch64-linux-gnu/libthread_db.so.1 \
        usr/lib/aarch64-linux-gnu/libicutu.so.74.2\
        usr/lib/aarch64-linux-gnu/libutil.so.1; do
      if [ -e $SNAPCRAFT_PRIME/$lib ]; then
         rm $SNAPCRAFT_PRIME/$lib
      fi
      done
apps:
  dia:
    command: usr/bin/dia
    extensions: [gnome]
    environment:
      DIA_BASE_PATH: "$SNAP/usr/local/share/dia"
      DIA_LIB_PATH: "$SNAP/usr/lib/"
      LD_LIBRARY_PATH:
         "$SNAP/usr/lib/aarch64-linux-gnu/libproxy:$SNAP/usr/lib/aarch64-linux-gnu:$
      DIA_UI_PATH: "$SNAP/usr/share/dia/ui"
      DIA_PYTHON_PATH: "$SNAP/usr/share/dia/python"
```

```
    DIA_XSLT_PATH: "$SNAP/usr/share/dia/xslt"
    DIA_SHEET_PATH: "$SNAP/usr/share/dia/sheets"
    DIA_SHAPE_PATH: "$SNAP/usr/share/dia/shapes"
  plugs:
    - x11
    - wayland
    - opengl
    - home
```

Bau des Dia-Snap

Nachdem die Yaml-Datei fertiggestellt war, war er bereit, den Dia-Snap zu bauen. Dazu gibt es zwei Schritte. Der erste Schritt besteht darin, den Erstellungsprozess zu initialisieren, der dann automatisch von Snapcraft verwaltet wird. Dieser Prozess wird mit dem folgenden Befehl im Stammverzeichnis des zu erstellenden Snap gestartet, in dem sich auch die Yaml-Datei befindet:

```
$ snapcraft init
```

Der vorliegende Befehl veranlasst die Installation einer .snap-Datei im Stammverzeichnis. Im Anschluss ist eine lokale Installation erforderlich, welche mittels des folgenden Befehls durchgeführt werden kann:

```
$ sudo snap install ./dia_0.1_arm64.snap --dangerous
```

Der Aufruf der Option –dangerous erlaubt den Systemzugriff und die Installation ohne Überprüfung.

Nach diesen Schritten war Dia schließlich als Snap verpackt und funktionierte wie jede andere Anwendung.

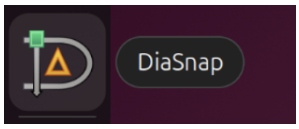

ABBILDUNG 3.7: Dia(Snap)

3.4.3 Verteilung an anderen Ubuntu Rechner

Der erste Schritt besteht darin, die gepackte Standalone-Datei dia.snap mit einem USB-Stick oder einer anderen Methode auf einen anderen Ubuntu-PC zu übertragen.

Zur Installation des Pakets wird das Programm snap benötigt, das mit folgendem Befehl installiert wird:

```
$ sudo apt install snap
```

Anschließend wird es durch Ausführen von den folgenden Befehl in dem PC installiert:

```
$ sudo snap install dia_0.1_arm64.snap --dangerous
```

3.4.4 Verteilung an Fedora Linux

Nach übertragung von die gepackte Standalone-Datei dia.snap sind folgende befehle benötigt zum installieren:

Installation des snapd-Daemons und Aktivierung der Unterstützung für *classic* Confinement.

```
$ sudo dnf install snapd
$ sudo ln -s /var/lib/snapd/snap /snap
```

Installation von Dia:

```
$ sudo snap install dia_0.1_arm64.snap --dangerous
```

3.4.5 Verteilung an Manjaro Linux

Manjaro hat den Befehl Snap standardmäßig installiert. Wenn dies jedoch nicht der Fall ist oder er entfernt wurde, kann er einfach mit dem folgenden Befehl installiert werden.

```
$ sudo pacman -S snapd
```

Nach der Installation ist die Aktivierung der systemd-Einheit erforderlich, die für die Verwaltung des Haupt-Snap-Sockets zuständig ist:

```
$ sudo systemctl enable --now snapd.socket
```

Aktivierung der Unterstützung für *classic* Confinement:

```
$ sudo ln -s /var/lib/snapd/snap /snap
```

Installation von Dia:

```
$ sudo snap install ./dia_0.1_arm64.snap --dangerous
```

Kapitel 4

Bewertung der Paketmanagersysteme anhand Relevenaten Eigenschaften

4.1 AppImage: Leichtgewichtig und portabel

AppImage verpackt Anwendungen als einzelne ausführbare Dateien. Dieser Ansatz gewährleistet die Portabilität und vermeidet Abhängigkeitskonflikte, indem wichtige Bibliotheken in das Paket aufgenommen werden.

Vorteile:

- Müssen nicht vom Benutzer installiert werden
- Benötigen keine Administrator-Rechte zur Ausführung
- Für einen Entwickler ist die Erstellung von AppImages recht einfach
- Sobald das AppImage erstellt ist, läuft es auf allen gängigen Linux-Distributionen

Nachteile:

- Eingeschränktes Sandboxing
- Keine Automatisierte Aktualisierung

Dia Speicherverbrauch: 10.7 MiB

In Bezug auf den Speicherbedarf erweist sich das AppImage-Format von Dia als dasjenige mit dem geringsten Bedarf unter den drei untersuchten Formaten. Dies lässt sich auf das schlank gestaltete Design zurückführen, wodurch sich das AppImage-Format insbesondere für Anwendungen eignet, bei denen die Größe und Einfachheit eine entscheidende Rolle spielen.

4.2 Flatpak: Sandboxing und Sicherheit

Flatpak legt den Schwerpunkt auf die Isolierung und Sicherheit von Anwendungen. Es verwendet ein System von Laufzeitumgebungen und Sandboxen, um ein konsistentes Verhalten über verschiedene Distributionen hinweg zu gewährleisten und gleichzeitig das Hostsystem vor

potenziell schädlicher Software zu schützen. Das modulare Design trennt die Anwendung von der Laufzeitumgebung und ermöglicht so effiziente Aktualisierungen und die gemeinsame Nutzung von Ressourcen.[11]

Vorteile:

- Starke Isolierung
- Verteilungsübergreifende Unterstützung
- Zuverlässige Leistung

Nachteile:

- Größere Anlaufzeit aufgrund von Laufzeitschichten
- Erfordert Flatpak-Laufzeiten
- Flatpaks nehmen in der Regel mehr Platz ein

Dia-Speicherverbrauch: 18.2 MiB

Das Flatpak-Paket für Dia bietet ein gutes Gleichgewicht zwischen Sicherheit und Ressourceneffizienz. Es ist etwas größer als AppImage, aber der zusätzliche Speicher ist ein Vorteil gegenüber der verbesserten Sandboxing- und Laufzeitflexibilität.

4.3 Snap: Eigenständig und konsistent

Snap ist das containerisierte Paketformat von Canonical. Im Gegensatz zu Flatpak sind Snap-Pakete vollständig in sich geschlossen, einschließlich aller erforderlichen Abhängigkeiten und Laufzeiten. Dieses Design gewährleistet eine konsistente Leistung über verschiedene Distributionen hinweg, kann aber zu größeren Paketen und einem höheren Speicherbedarf führen.

Vorteile:

- Funktioniert auf jedem Linux-System mit Snapd
- Enthält alle Abhängigkeiten
- Starke Isolierung

Nachteile:

- Größere Speichernutzung aufgrund der gebündelten Abhängigkeiten
- Mögliche Startverzögerung
- Snaps lassen sich beim ersten Start nur sehr langsam öffnen.
- Alle Abhängig

Dia Speicherverbrauch:(keine Laufzeitumgebung) 72.3 MiB
Dia Speicherverbrauch:(gnome Laufzeitumgebung) 57.7 MiB

Erstaunlicherweise verbrauchte das Snap-Paket von Dia deutlich mehr Speicher als die anderen Formate. Während Snap normalerweise von Kompression und Deduplikation profitiert, könnte dieses Missverhältnis auf redundante Abhängigkeiten oder Ineffizienzen im Paketierungsprozess für diese spezielle Anwendung zurückzuführen sein. Durch die Verwendung der Gnome-Laufzeitumgebung wurde das Paket jedoch auf etwa 20 MiB reduziert. In Bezugnahme auf die beiden anderen Methoden ist festzustellen, dass die Größe jedoch nach wie vor als relativ groß zu bezeichnen ist.

Merkmal	**AppImage**	**Flatpak**	**Snap(gnome)**
Speicherverwendung	10.7 MiB	18.2 MiB	57.7 MiB
Handhabung von Abhängigkeiten	Minimal	Gemeinsame Laufzeit	Gebündelte Abhängigkeiten
Sandboxing	Nein	Ja	Ja
Übertragbarkeit	Sehr hoch	Hoch	Hoch
Größe der Laufzeit	Keine	Gemeinsame Laufzeit: 600+ MiB	Gemeinsame Laufzeit: 421MiB
Einfache Aktualisierung	Gesamte Datei ersetzt	Delta-Aktualisierungen	Delta-Aktualisierungen
Komplexität aufbauen	Einfach	Mäßig	Hoch

TABELLE 4.1: Vergleich von AppImage, Flatpak und Snap für die Verpackung von Dia

Kapitel 5

Resümee

Moderne Paketmanagersysteme wie Flatpak, Snap und AppImage haben die Softwareverteilung unter Linux grundlegend verändert. Sie lösen das seit langem bestehende Problem der Querverteilung, Kompatibilität und vereinfachen gleichzeitig den Entwicklungsprozess für Softwareentwickler. In dieser Arbeit wurde untersucht, welche technischen Unterschiede zwischen diesen Paketmanagersystemen bestehen, wie sie in der Praxis eingesetzt werden und welche Auswirkungen sie auf Entwickler und Nutzer haben.

5.1 Wichtige Beiträge

In dieser Arbeit diente Dia, eine Diagrammeditor-Anwendung, als Gegenstand für das Erstellen und Testen von Paketen in jedem der drei Formate. Der Paketierungsprozess wurde gründlich untersucht, um die Stärken und Grenzen der einzelnen Systeme aufzudecken.

5.1.1 Technische Umsetzung

Das **AppImage-Format** ermöglichte Einfachheit und Portabilität, da Dia als selbständige ausführbare Datei verpackt war, die ohne Installation ausgeführt werden konnte. Der Prozess war relativ einfach, aber das Fehlen einer nativen Sandbox stellte ein potenzielles Sicherheitsrisiko dar.

Flatpak ermöglichte eine effiziente Lösung für die Verteilung von Anwendungen, indem es gemeinsam genutzte Laufzeiten, wie org.gnome.Platform, nutzte. Der Prozess zeigte, wie gemeinsame Abhängigkeiten die Redundanz minimieren und die Paketgröße reduzieren, obwohl Laufzeitanforderungen einen gewissen Overhead verursachen können.

Snap packte Dia als eigenständige Anwendung und bündelte alle Abhängigkeiten. Dies gewährleistete die Kompatibilität zwischen verschiedenen Systemen, führte jedoch zu einer größeren Paketgröße im Vergleich zu den anderen Formaten. Die robusten Confinement-Niveau von Snap verbesserten die Sicherheit.

5.1.2 Herausforderungen und Lösungen

In dieser Arbeit wurden mehrere Herausforderungen bei der Paketierung von Dia identifiziert und behandelt, z. B. die Lösung von Abhängigkeitskonflikten, die Verwaltung von Sandbox-Berechtigungen und das Debugging

von Laufzeitproblemen. Für jedes Format wurden praktische Lösungen vorgeschlagen. Diese können auch für andere Entwickler nützlich sein.

5.2 Zukünftige Implikationen

Die Entwicklung von universellen Paketmanagementsystemen ist ein Indikator für die steigende Nachfrage nach distributionsunabhängigen Softwarelösungen. Diese Systeme haben nicht nur den Vorteil, dass sie den Entwicklungs- und Verteilungsprozess vereinfachen, sondern machen Linux auch für Benutzer und Entwickler leichter zugänglich.

Allerdings bestehen trotzdem einige Herausforderungen:

5.2.1 Verbesserung von Paketgröße und Leistung

Obwohl Flatpak und Snap Fortschritte bei der Reduzierung der Update-Größen durch Delta-Updates gemacht haben, sehen viele Nutzer die Gesamtgröße der Pakete weiterhin als problematisch an. Es ist zu erwarten, dass künftige Innovationen auf die Optimierung gemeinsam genutzter Laufzeiten und die Entwicklung verbesserter Komprimierungstechniken abzielen werden.

5.2.2 Gleichgewicht zwischen Sicherheit und Benutzerfreundlichkeit

Die Implementierung von Portalen in Flatpak sowie die Arten der Beschränkungen von Snap stellen einen signifikanten Fortschritt in puncto Sicherheit dar. Allerdings können diese Mechanismen mitunter die Funktionalität einschränken. Das optimale Gleichgewicht zwischen Sicherheit und Benutzerfreundlichkeit stellt nach wie vor einen Bereich dar, der einer weiteren Optimierung bedarf.

5.2.3 Ausweitung der Akzeptanz in der Gemeinschaft

Der Erfolg dieser Systeme ist maßgeblich von der Unterstützung durch die Gemeinschaft sowie durch die Entwicklerinnen und Entwickler abhängig. Eine Optimierung der Dokumentation, eine Reduzierung der Lernkurve für die Paketierung sowie die Behebung von Kritikpunkten, beispielsweise hinsichtlich des zentralisierten Charakters von Snap, könnten zu einer erhöhten Akzeptanz beitragen.

5.3 Schlussfolgerung

Die Untersuchung moderner Paketmanagersysteme legt dar, dass diese eine wesentliche Rolle bei der Fortentwicklung des Linux-Ökosystems einnehmen. Diesbezüglich sind nicht nur technische Innovationen von Relevanz, sondern auch die Frage, wie eine Vereinbarkeit von Softwarezugänglichkeit und Sicherheitsstandards gewährleistet werden kann.

Die Integration dieser Systeme in den Alltag der Nutzer sowie deren Weiterentwicklung stellen die zentralen Aufgaben der kommenden Jahre dar. Dabei wird die Ausgewogenheit zwischen technologischer Flexibilität, Sicherheit und Nutzerfreundlichkeit von entscheidender Bedeutung sein. Der Erfolg solcher Technologien ist letztlich von der Anpassungsfähigkeit des Linux-Ökosystems abhängig, welches gleichzeitig neue Standards für die globale Softwareverteilung setzt.

5.4 Ausblick auf die Zukunft

In Zukunft möchte ich an diesem Projekt weiterarbeiten. Ich bin daran interessiert die verpackte Dia Anwendung in den jeweiligen offiziellen Stores zu veröffentlichen, was ich in diesem Zeitraum nicht geschafft habe.

Ich freue mich darauf, mein Wissen mit der Open-Source Community zu teilen und die Möglichkeit zu nutzen, Dia mit den entsprechenden Verpackungsmethoden anderen leichter zur Verfügung zu stellen.

Literatur

[1] Roderick W. Smith. *Linux Essentials*. John Wiley und Sons Inc., 2012, S. 85–86. ISBN: 9781118106792.

[2] Aaron Weber Ellen Siever Stephen Figgins. *Linux in a nutshell*. O'Reilly, 2003, S. 36.

[3] Atanas Georgiev Rusev. *Manjaro Linux User Guide*. Packt Publishing, 2023, S. 135–136. ISBN: 9781803239712.

[4] *Dia Documentation*. URL: https://gitlab.gnome.org/GNOME/dia/-/blob/master/BUILDING.md.

[5] Atanas Georgiev Rusev. *Manjaro Linux User Guide*. Packt Publishing, 2023, S. 136. ISBN: 9781803239712.

[6] ALEX. CALLEJAS. *Fedora Linux System Administration: Install, manage, and secure your Fedora Linux environments*. Packt, 2023, S. 189. ISBN: 9781804618400.

[7] *Appimage Documentation: Introduction to Packaging*. URL: https://docs.appimage.org/packaging-guide/introduction.html.

[8] Jürg Rechsteiner. *Linux für Einsteiger und Fortgeschrittene*. epubli, 2024.

[9] Atanas Georgiev Rusev. *Manjaro Linux User Guide*. Packt Publishing, 2023, S. 196. ISBN: 9781803239712.

[10] URL: https://flatpak-test.readthedocs.io/en/latest/introduction.html.

[11] Richard Blum Christine Bresnahan. *Mastering Linux System Administration*. Wiley, 2021, S. 397–398.

[12] Flatpak Project. *Flatpak Documentation: Building Introduction*. 2024. URL: https://docs.flatpak.org/en/latest/building.html.

[13] Flatpak Team. „Flatpak test". In: (2016). URL: https://flatpak-test.readthedocs.io/en/latest/introduction.html.

[14] ALEX. CALLEJAS. *Fedora Linux System Administration: Install, manage, and secure your Fedora Linux environments*. Packt, 2023, S. 215–216. ISBN: 9781804618400.

[15] Richard Blum Christine Bresnahan. *Mastering Linux System Administration*. Wiley, 2021, S. 397.

[16] *Snapcraft Documentation: Snapcraft extensions*. URL: https://snapcraft.io/docs/snapcraft-extensions.