Custom Package Framework

An der

Montessori Schule Huckepack e. V.

Vorgelegt von:

Tendsin Mende

Pohlandstraße 5

01309 Dresden

Abgabetermin:

27.02.2016

Betreuer:

Herr Diplom. Lehrer Gerd Bobe

Montessori Schule Huckepack e. V.

Inhaltsverzeichnis

1 Einführung

1.1 Die Situation

„Schön! Ich habe ein Linux, aber wie bearbeite ich ein Bild?“

An diesem Problem möchte ich mit meiner Arbeit ansetzen. Ich möchte ein- und Umsteigern das Suchen von Programmen erleichtern. Ich sehe drei Hauptprobleme für den Neueinstieg.

1.2 Grund 1

Auch wenn es die Programme von Windows und MacOSx nicht gibt, so ist die Auswahl an Programmen zu groß. Als Beispiel: Es gibt kein offizielles Adobe: Photoshop für Linux[1]. Es gibt aber Gimp, Krita, MyPaint, und viele mehr. Welches davon soll ich nehmen? Aus meiner Erfahrung kann ich sagen: „Krita für malen und Texturbearbeitung und Gimp für Photos“. Aber diese Erfahrung hat der Normalverbraucher nicht. Meine Lösung ist einfach: Ich gebe dem Nutzer nur Gimp und Krita und eine Beschreibung mit meiner Erfahrung.

1.3 Grund 2

Das nächste Problem ist recht trivial. Wie installiere ich Software?

Die meisten Umsteiger kommen von Windows. Sie erwarten, dass man ins Internet geht, den richtigen „Installer“ herunterlädt und dann installiert. Auf Linux kann man das Installieren aber eher mit einem Appstore vergleichen. Man wählt sein Paket (Programm) und lädt es sich von einem Spiegel-Server herunter. Das Paket weiß selbst welche anderen Programme es benötigt und installiert diese nach. Der Vorteil dieser Methode ist, dass sich viele Programme einzelne Unterprogramme teilen können. Außerdem lässt sich das System so einfach mit dem Server abgleichen. Das Aktualisieren der Programme ist einfacher.

2 Vorüberlegungen

2.1 Ziel

Meine ersten Überlegungen galten zwangsläufig meinem Ziel. Wie sollte mein Programm aussehen um einen möglichst einfachen einstieg in Linux zu gewährleisten?

[1] Vgl. https://helpx.adobe.com/photoshop/system-requirements.html, (14.08.2016, 13:57)

Ich kam zu dem naheliegenden Schluss, dass es das beste sei eine einfache Oberfläche zu konstruieren. Es entstanden die ersten Ideen.

Meine fertige Idee bestand aus einem Rahmen, welcher in drei Teile gegliedert war. Der Programmauswahl, ein Dokumentationsbrowser und einem Informationssystem. Die Programmauswahl sollte Icons für verschiedene Programm-arten bekommen (z. B. „Video und Foto“, „Musik“, „Spiele“). Durch das Anklicken der Icons sollte man in eine Feinauswahlen kommen, bis man nach maximal drei Verzeichnissen eine Programmauswahl präsentiert bekommt.

Der Dokumentationsbrowser soll die Aufgabe übernehmen, Dokumentation zugänglich zu machen. Dabei denke ich primär an Schul- oder firmeninterne Dokumentation.

Der Informationsbrowser wiederum soll über neue Entwicklungen informieren. Es bietet sich an Blog-Webseiten oder Nachrichten an dieser Stelle zu laden.

Ein nicht zu unterschätzender Teil ist die Administration eines solchen Programms. Ich weiß aus Erfahrung meines ersten größeren Programms[2], dass man ein gutes Konfigurations-System braucht. Bei CPF (kurz für: Custom Packaging Framework) würde außerdem noch ein weiteres Problem hinzu kommen. Da es sich um die Installation von Programmen handelt, muss die Sicherheit des System gewährleistet sein. Das heißt es muss genau geregelt werden wie man installiert und welche Programme welchen zugriff auf Dateien haben.

2.2 Konfiguration

2.2.1 Programm eigene Konfiguration

Die Programm eigene Konfiguration umschließt die Art, wie CPF reagieren soll. Dazu sollte gehören:

- Umgang mit der Datenbank
  - Speicherung von Daten
  - Abrufen von Daten
- Umgang mit dem Internet
  - Laden der Webseiten in Dokumentation und Info-System
  - Herunterladen von Programmen
  - Abgleichen der Datenbank mit online Repositorium
- Verhalten während der Installation

[2]siehe: https://github.com/SiebenCorgie/Beta-Launcher , 18.08.2016, 18:30

  - Automatisierungsgrad während der Installation
  - Passwort-abfrage (welcher Benutzer fragt?)
- Massenaufträge
  - Grafische Rückmeldung

Die verschiedenen Optionen werden in verschiedenen Registerkarten eines Konfigurationsfensters geändert. Das ist erfahrungsgemäß die übersichtlichste Art.

2.2.2 Datenbank

Die Datenbank soll aus einfachen Datensätzen mit verschiedenen Daten beruhen. zu diesen Daten soll gehören:

- Name des Programms
- Entwicklerbeschreibung (Kurzbeschreibung)
- Eigene Beschreibung (lange Beschreibung)
- Programm Typ (Grafik, Video, Spiel, etc.) (Überkategorie)
- Unterkategory
- Programm Website
- Pfad zu Screenshot

2.2.3 Datenbank Management

Die Datenbank soll außerdem erweiterbar sein. Dazu muss auch ein eigenes Tool geschrieben werden. Die Erweiterung wird aus einer Eingabemaske bestehen sowie der Option die lokale Datenbank in das Repositorium hoch zu laden.

2.3 Installation

2.3.1 Visualisierung

Das Installieren von Software soll mit minimalem visuellen eindrücken auskommen. Dabei möchte ich nur den Fortschritt, und den Namen der Aktion anzeigen. Die Werte sollen im unteren Teil des Programms angezeigt werden.

2.3.2 Installationsablauf

Die Installation soll wie folgt ablaufen:

1. Programm Auswahl
2. Installations-Start
3. Passwort-abfrage (Benutzer wird aus Konfigurationsdateien gelesen)
4. Bestätigung
5. Installation
6. Erfolgs/Misserfolgsnachricht

Bei einem Misserfolg soll nach Möglichkeit der Fehler angezeigt werden. Insofern benötigt soll man auch den Lockbuch-Eintrag sehen können.

2.3.3 Massenaufträge

Das Programm soll Massenaufträge abarbeiten können. Dies soll vor allem Administratoren die Arbeit ersparen. Bei einem Massenauftrag werden viele Programme mit einem mal installiert.

2.3.4 Programmauswahl

Die Programme sollen wie in 2.1 beschrieben durch fein Filtern der Optionen ausgewählt werden. Ich werden versuchen die Beschreibungen möglichst neutral zuhalten. Damit soll die Entscheidung des Nutzers möglichst neutral bleiben, um zum gewünschten Programm zu gelangen.

2.4 Web-Integration

Die Webintegration soll über PyWebKit[3] erfolgen. Das ist eine von Apple entwickelte Umgebung, um relativ einfach Browserfähigkeiten in Gtk-Programme einzubauen. Ich werde es verwenden um einfache Webseiten anzuzeigen. Die Lizenzierung von WebKit[4] ermöglicht es mir das Programm kommerziell oder nicht-kommerziell zu entwickeln und zu veröffentlichen.

[3]siehe: https://webkit.org/ , 14.08.2016, 20:18
[4]https://webkit.org/licensing-webkit/ , 14.08.2016 , 20:21

2.5 Programmiersprache

Als Programmiersprache nehme ich Python.Es gibt verschiedene Gründe dafür.

Der Erste ist, dass ich schon seit Sommer 2015 Programme mit Python schreibe. Ich kann es relativ „flüssig“ schreiben. Diese Vorkenntnis ist nötig um in der gegebenen Zeit ein so komplexes Programm zuschreiben.

Der zweite Grund ist, dass Python eine interpretiere[5] Sprache ist. Der Vorteil ist, dass ein Python-Programm bei einem Programmfehler nicht abstürzt. Es führt in meinem Fall den Befehl nicht aus und kehrt in die Warte-Position zurück[6]. Dadurch ist das potenzielle Installieren fehlerhafter Software, oder frustrierende abstürze des Programms geringer. Ein Nachteil ist, dass man Programmfehler erst bei der Ausführung findet. Das Testen wird dadurch aufwendiger.

Der Dritte Grund ist Pythons Anbindung an das GTK+ Projekt. Wie ich im nächsten abschnitt beschreiben werde nutze ich das GTK+ um meine grafische Umsetzung zu realisieren. Bei einem so grundlegenden Element ist es wichtig, dass die Anbindung an die Programmiersprache gut ist. Dies ist bei Python glücklicherweise gegeben.

2.6 Interface-Toolkit

Das Interface-Toolkit[7] ist auf Linux zu Verständigung zwischen dem Display-Server (z.B. X11 oder Wayland) und dem Benutzer zuständig. Es stellt verschiedene Elemente bereit, die zusammen das fertige Programm-Interface ergeben (siehe Abb.1)[8].

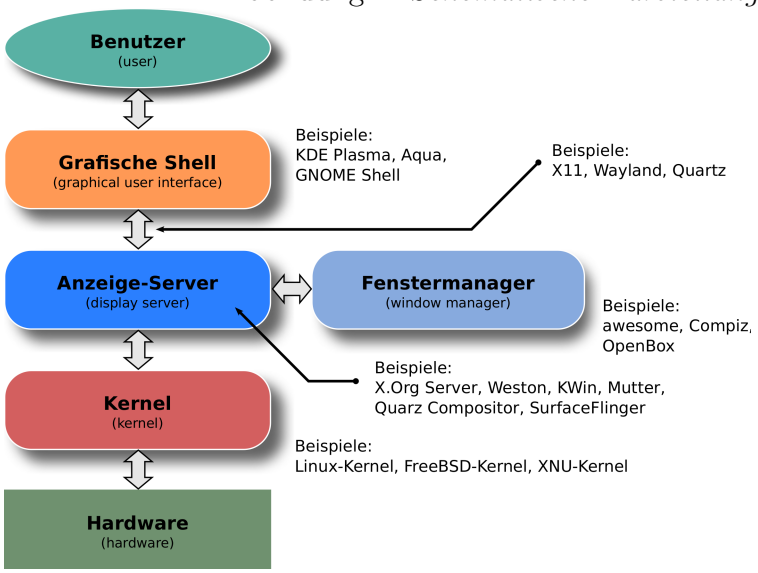
[5]Siehe: https://de.wikipedia.org/wiki/Interpreter

[6]Die Charackteristik eines Gtk+ Interfaces wird Später noch geklärt.

[7]kurze Erklärung: https://de.wikipedia.org/wiki/Toolkit

[8]Quelle: https://upload.wikimedia.org/wikipedia/commons/thumb/2/23/Schema_der_Schichten_der_grafischen_B
Schema_der_Schichten_der_grafischen_Benutzeroberfl%C3%A4che.svg.png , Shmuel Csaba Otto Traian, 29.10.2013

Abbildung 1: *Schematische Darstellung der Beziehungen*



Ich habe mich für GTK+[9] entscheiden. Der wichtigste Grund wurde schon in „2.5 Programmiersprache“ angeführt. GTK+ hat eine exzellente Pythonanbindung (ab jetzt API , kurz für Application-Programming-Interface). Allerdings macht sich GTK+ durch weitere Vorteile prädestiniert für den Einsatz in meinem Programm.

Ein weiterer großer Vorteil ist die Lizenzierung. Gtk+ ist unter LGPL[10] Lizenziert[11]. Diese erlaubt es mir GTK ohne Restriktionen einzusetzen. Die LGPL Lizenz ist eine der am weitesten verbreiteten Lizenzen in der OpenSource-Welt. Der Erfolg durch die Lizenz und die Qualität lässt sich auf Linux leicht erkennen. Es gibt sehr viele Programme die mit dem GTK erstellt wurden. Des weiteren gibt es sogar eine ganze Desktop-Umgebung die auf GTK basiert. Genannt „Gnome“[12], welche ich selbst benutze. Das GTK Projekt ist eines der ältesten[13] und erfolgreichsten Linux-Projekte im grafischen Segment. Mit dem Erfolg kommt eine weite Verbreitung im Linux-Anwenderbereich. Der kommt mir insofern zugute, als das es auf fast allen Distributionen die benötigten Pakete für mein Programm gibt.

Der letzte Grund für das GTK+ ist meine Erfahrung. Wie in 2.5 erwähnt habe ich schon einige Erfahrung auf dem Gebiet von Python. Dazu gehört es auch, dass ich ab Ostern 2016 ein Programm[14] zum Organisieren und Installieren der UnrealEngine 4 auf Linux geschrieben habe. Dieses war auch mit GTK erstellt worden. Ich habe während des Programmierens viel Erfahrung gesammelt. Damals habe ich mehrere Toolkits miteinander verglichen (z.B. Qt und wxWidgets). Am Ende habe ich mich für das Gtk entscheiden.

[9]Englisch: http://www.gtk.org/ ,18.08.2016, 21:25
[10]Englisch: http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html , 18.08.2016 , 21:32
[11]Englisch: http://www.gtk.org/ , 18.08.2016, 21:32
[12]Englische Referenz: https://www.gnome.org/
[13]https://www.gimp.org/about/ancient_history.html , 08.18.2016 , 21:47
[14]siehe: https://github.com/SiebenCorgie/Beta-Launcher

Das Ergebnis ist ein gefestigtes Anfängerwissen über ein Toolkit welches ich mir bewusst gesucht habe.

2.7 Unterstützung

Ich plane die Unterstützung des .deb Formates und des .tar.xz Formates. Das heist es gibt Unterstützung von Debian und dessen Distributionen sowie Arch-Linux und dessen Distributionen.

Dabei ist zu beachten das Ubuntu eine andere „naming-convention“ hat als Debian, allerdings trotzdem auf dem .deb Format beruht. Deshalb werde ich in meiner Programmdatenbank zwei verschiedene Speicherplätze für Debian und Ubuntu basierte Systeme einrichten.

In der Linux-Welt gibt es noch ein anderes, sehr weit verbreitetes Format: .rpm[15]. Mit diesem Format habe ich allerdings keine Erfahrung. Deshalb werde ich es nicht benutzen. Es wäre möglich sich mit dem Format zu beschäftigen, allerdings hängt es sehr stark von der gewählten Distribution ab wie man das Paket erstellt. Deshalb habe ich mich entschieden das Format vollkommen aus zu lassen.

2.8 Arbeitsweise

2.8.1 Schreiben

Ich werde das die Dokumentation mit Lyx[16] schreiben. Das ist ein grafisches Programm zum LATEX-Framework[17]. Der Grund dafür ist vor allem die gute Linux-Unterstützung sowie das exzellente Schriftbild welches durch das LATEX Programm erstellt wird. Des weiteren ist LATEX als Dokumentensprache in wissenschaftlichen Arbeiten weit verbreitet. Die Arbeit damit stellt also auch eine gewissen Vorbereitung für mich dar.

2.8.2 Git-Hub

Das Programm sowie die Dokumentation kommen in eine Programm-Ordnerstruktur. Diese soll mit einem Git-Hub Repositorium abgeglichen werden. Dadurch erreiche ich maximale Transparenz was das Programm und dessen Fortschritt angeht. Außerdem ist das bearbeiten von Programmen über Git-Hub eine sehr weit verbreitete Praxis in der Linux-Welt.

[15] Kurze Erklärung: https://en.wikipedia.org/wiki/RPM_Package_Manager , 01.09.2016, 16:49
[16] http://www.lyx.org/ , 01.09.2016, 17:37
[17] http://www.latex-project.org/ , 01.09.2016, 17:35

Git-Hub ist eine Online-Umgebung bei der man Programme hochladen kann. Es hebt sich von normalen Web-Hosts in sofern ab, als das man als Administrator des Repositoriums Änderungen anderer annehmen und prüfen kann. Außerdem gibt es einen sehr ausgeklügelten Mechanismus um lokale Versionen eines Programms mit der Onlineversion zu synchronisieren[18].

3 Dokumentation der Arbeit

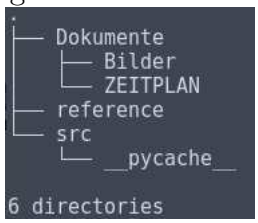
3.1 Voranmerkung

Die Dokumentation ist in der Reihenfolge in der ich das Programm geschrieben habe. Es ist außerdem gut eine Version des Programm-Listings griffbereit zu haben. Ich werde hin und wieder indirekt Bezug auf einzelne Stellen nehmen. Insofern erforderlich baue ich auch Listing-Stellen in den Text ein.

3.2 Erstellung des Programms

Das Programm selbst lasse ich zuerst durch den Python-GTK-Wizard von Anjuta erstellen. Das ist schnell durch den Dialog „Create New Project“ mit den Einstellung für ein Python-GTK-Projekt getan. Dieser Wizard erstellt in einem Ordner die Typische Ordnerstruktur eines Linux-Programms. Nach dem Hinzufügen meiner eigenen Ordner für Dokumentation und Referenzen haben ich die Ordnerstruktur aus Abb. 2.

Abbildung 2: *Grund Ordnerstruktur*



Dieser Ordnerstruktur soll mit einem Git-Hub Repositorium abgeglichen werden. Dazu kreiere ich zuerst ein leeres Repositorium (ab jetzt Repo.) auf Git-Hub mit meinem Benutzer. Danach klone ich es auf meinen Computer. In den zurzeit leeren Ordner kopiere ich das gerade erstellte Projekt. Danach synchronisiere ich mithilfe der folgenden Kommandos:

[18]Mehr Informationen über Git-Hub: https://de.wikipedia.org/wiki/GitHub , 01.09.2016, 17:46

```
# Dateien und Ordner zu lokalen Repo hinzufügen
git add *
# Dateien lokal Synchronisieren und Nachricht hinterlassen
git commit -a
# Lokale neue Version online synchronisieren
git push
```

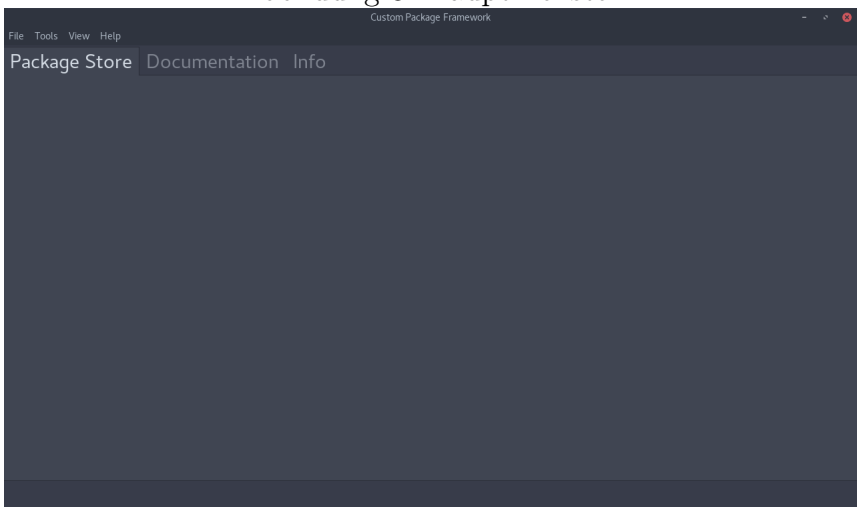
Jetzt ist die Grundstruktur online Hinterlegt. Ich kann mir jederzeit eine Version der Struktur herunterladen, bearbeiten und die neue Version hochladen. Der erste „commit“ hat die Nachricht: „Initial commit“.

3.3 Erstellung der Grafischen Oberfläche: Hauptfenster

In Anjuta gibt es „Glade“ den UI-Editor für mein GTK-Interface. Mit ihm kann ich mithilfe verschiedener vorgegebener UI-Elementen mein Hauptfenster erstellen. Die einzelnen Schritte beinhalten immer das einfügen des Widgets, die Namensgebung sowie die Einstellung gewünschter Parameter.

Ich entscheide mich für eine traditionelle „Taskbar“ am oberen Fensterrand, sowie einem vertikalen „Notebook“. In dem Notbook werden auf der ersten Position später die verschiedenen Programmkategorien erscheinen. Auf der zweiten und dritten Position ist der „Dokumentation-Browser“ und der „Informations-Browser“. Das fertige Ergebnis sieht wie in Abb. 3 aus.

Abbildung 3: Haupt-Fenster



3.4 Browserfenster

Die Browserfenster werden beim Programmstart eingebunden. Dazu erstelle ich in dem Fenster von „Documentation“ bzw. „Info“ ein WebKit-Browser. Dieser lädt die „Home“

Url.

Wenn die Interaktion mit dem Internet in den Einstellungen deaktiviert ist wird WebKit garnicht eingebunden und stattdessen ein Fehler-Icon angezeigt.

3.5 Konfiguration

3.5.1 Konfigurationsdatei

Zuerst erstelle ich mir eine eigene Konfigurationsdatei. In ihre werden alle Konfigurationen gespeichert die das Programm über mehrere Läufe hinweg behalten soll. Die Datei ist in Überabschnitte geteilt. Deren Namen sind in eckigen Klammern eingefasst. Es folgen die Namen. Nach einem Gleichheitszeichen kommt der Wert. Eine Überkategorie mit Wertenamen und Wert sähe also wie folgt aus:

```
[Ueberkategorie]
Name = Wert
```

Diese Syntax kann von einem Modul in Python gelesen werten. Dadurch ist es für mich einfachere die Werte auszulesen und zu ändern.

3.5.2 Lesen und Schreiben

Allgemeines Wie schon angekündigt nehme ich zum lesen und schreiben der Werte ein Modul. Dieses heißt „Configparser“[19]. Mit der in [8] beschriebenen API kann ich nun Dateien ein und auslesen, sowie Einträge ändern. Der Pfad zur Konfigurationsdatei ist im Programmierstatus noch lokal direkt im Verzeichnis. Im installierte zustand wird die Konfigurationsdatei immer im Home-Verzeichnis des Nutzers liegen.

Für die Konfiguration legen ich eine neue Pythondatei mit dem Namen CPFConfig an. Diese importiere ich in das Hauptprogramm unter dem Pseudonym „conf“. In der neuen Datei erstelle ich die Funktionen:

1. set_entry (auslesen eines Wertes)
2. get_entry (einlesen eines Wertes)
3. load_config (laden der Werte des Konfigurationsmenüs)
4. save_config (schreiben der Werte des Konfigurationsmenüs)

Außerdem kommen vor die Funktionen noch die Befehle um die Konfigurationsdatei einzulesen. Dabei wird die Datei in die Variable „config“ als ConfigParser-Objekt gespeichert.

[19]Dokumentation: https://docs.python.org/3.5/library/configparser.html , 01.09.2016, 18:35

Damit kann ich auf „config“ alle Funktionen des ConfigParser Objekts anwenden. Danach lese ich mit dem Befehl „read“ die Konfigurationsdatei ein. Dadurch das die Befehle am Anfang der Datei stehen werden sie ausgeführt sobald dieses Modul gestartet wird. Erst danach werden die Funktionen gelesen und bei Aufrufen ausgeführt.

Die Funktion set_entry Die Funktion nimmt Werte für die Variablen „section“, „entry“ und „newentry“ entgegen. Danach wendet sie „set“ auf das „config“ Objekt an. „Set“ bekommt als Argumente die Kategorie des neuen Wertes („section“) den Namen („entry“) und den neuen Wert („newentry“).

Danach schreibt sie die neuen Werte in die Konfigurationsdatei.

Die Funktion get_entry Die Funktion nimmt zwei Werte entgegen: „section“ und „entry“. Ähnlich wie bei set_entry werden die werte auf das „conf“ Objekt angewendet um mit den Argumenten „section“ und „entry“ den korrespondierenden Wert wieder zugeben. Dabei steht diese Anweisung in der „return“ Anweisung. Dies sorgt dafür, dass der Aufrufer der Funktion den Wert in der Konfigurationsdatei zurück bekommt. Dieser Wert wird meist in einer Variable gespeichert.

Die Funktion load_config Die Funktion ist dazu da den Inhalt des Konfigurations-Dialogs fest zu legen. Sie nimmt als Argument nur den Konstruktor des GTK-Fensters (builder). Dadurch kann man in der Funktion auch das grafische Interface aus der Klasse „GUI“ bearbeiten. Dabei wird für jedes Widget der Wert aus der Konfigurationsdatei geladen. Es handelt sich in fast allen fällen um einen Text der geladen wird. Das ist relativ einfach:

Zuerst legt man eine Variable mit dem Objekt an. Anschließend wird der alte Wert des Textes mit dem neuen überschrieben. Der neue Wert wird über die get_entry Funktion aus der Konfigurationsdatei gelesen.

Es gibt allerdings auch zwei Sonderfälle. Einmal Verzeichnis-Auswahlen und eine „Ja/Nein“-Auswahl durch eine „Tick-Box“.

Bei der Verzeichnis-Auswahl liest das Programm den gespeicherten Pfad aus der Konfigurationsdatei. Danach wird die Datei über „set_filename“ ausgewählt.

Bei den Tickboxen hat man nur die Auswahl zwischen gesetztem Hacken oder leerem Feld. Deshalb liest das Programm die Konfigurationsdatei ein. Wenn an der stelle „True“ gespeichert ist wird der Hacken aktiviert. Ansonsten wird er deaktiviert.

Die Funktion save_config Diese Funktion Ist die Umkehrung der „load_config“ Funktion.

Sie liest über den „Builder“ alle Texte ein und speichert sie an ihrem Platz in der Konfigurationsdatei.

Texte werden dabei eingelesen, indem das Objekt als Variable gespeichert wird. Danach wird dessen Textobjekt ausgelesen und der Wert gespeichert.

Bei der Auswahl eines Verzeichnisses gibt es ein eigenes Widget. Bei diesem erfährt man den Pfad zum gewählten Verzeichnis durch die Methode „get_filename“. Diese gibt den Pfad als String aus.

Wenn keine Veränderung vorgenommen wurde ist allerdings der Wert der von „get_filename“ ausgeben wird „None“. Um zu verhindern, dass in der Konfigurationsdatei „None“ als Pfad gespeichert wird, liest das Programm den Pfad ein. Wenn der Pfad „None“ ist wird die Konfigurationsdatei nicht geändert. Ansonsten wird der neue Wert von „get_filename“ eingespeichert.

Bei der Tick-Box hat man nur die Möglichkeit einen Hacken zusetzen, oder ihn nicht zusetzen. Dabei ist die Schwierigkeit nur, dass wenn der Hacken gesetzt wird der String „True“ eingespeichert wird. Wenn er nicht gesetzt ist, wird „False“ eingespeichert.

3.5.3 Konfigurationsmenu

Das grafische Menü habe ich nach den Kategorien der Konfigurationsdatei gegliedert:

- Internet
- Datenbank
- Installation
- MassInstall

In den Kategorien gibt es eine Zwei-Spalten-Liste. Jeweils mit links der Beschreibung der Einstellung und rechts dem Wert. Also Text oder Auswahl-Dialog.

Auf der untersten Leiste liegen die Knöpfe zum Speichern und Verwerfen.

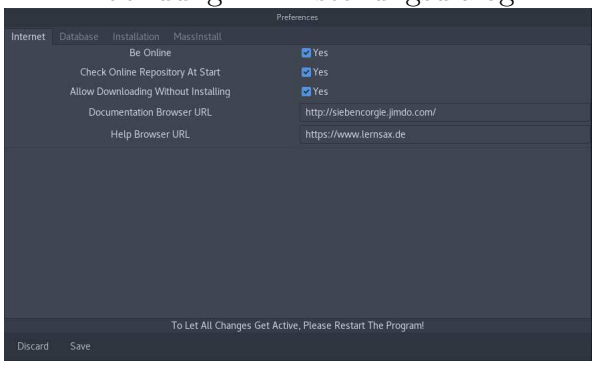
3.6 Datenbank

3.6.1 Technisches über die Datenbank

Die Datenbank schreibe ich über das Modul „sqlite3“. Das ist eine einfachere Version der beliebten SQL-Datenbank mit Pythonitegration[20]. Mit diesem Modul kreiert oder läd man eine Datenbank-Datei. Diese kann man anschließend über der „Curser“ also eine Art Pfeil manipulieren. Nach der Manipulation werden die Änderungen ähnlich wie bei Git-Hub gesammelt und die Datei mit den Änderungen aktualisiert.

[20]https://docs.python.org/2/library/sqlite3.html, 06.09.2016, 19:06

Abbildung 4: Einstellungsdialog



3.6.2 Datenbankiteraktion

Die Interaktion besteht aus mehreren Aktionen:

1. Erstellen einer Datenbank und einer Liste
2. Erstellen neuer Einträge
3. Lesen der Einträge nach verschiedenen Aspekten
4. Löschen einzelner Einträge

Die Datenbank enthält nur eine Liste: „CPFDB“. In dieser Liste steht:

1. Name
2. Kurzbeschreibung
3. Lange Beschreibung
4. Name in den Ubuntu Repositorien
5. Name in den Debian Repositorien
6. Name in den Arch Repositorien
7. Pfad zu einem Bildschirmfoto des Programms
8. Pfad zu einem Symbol
9. Überkategorie
10. Unterkategorie
11. Webseiten URL des Entwicklers

3.6.3 Erstellen einer neuen Datenbank

Die Datenbank im Dateiensystem umfasst nur eine Datei und zwei Ordner. Die Datei ist die Datenbank und die Ordner beinhalten Symbole und Screenshots der Programme.

Beim Erstellen werden die Ordner sowie die Datei erstellt. Dabei kann bei bedarf die neue Datenbank direkt in die Konfigurationsdatei eingetragen und damit aktiviert werden. Der angegebene Name in dem Erstellungs-Dialog entspricht dem Name der erstellten Datenbank.

Es ist zu beachten, dass Namen auf Linux nicht die Art der Datei beinhalten müssen. Der Name: „Datenbank.sh“ ist genauso zulässig wie „Datenbank.txt“ oder einfach nur „Datenbank“.

3.6.4 Erstellen neuer Einträge

Das Erstellen neuer Einträge ist denkbar einfach. Man bekommt den Dialog präsentiert. In dem Dialog kann man für jeden Eintrag eines Datensatzes in der Datenbank einen Wert angeben. Vom Name, bis zur URL der Entwickler-Homepage.

Eine kleine Ausnahme machen hier die Auswahl des Screenshots und des Symbols. Bei der Aktivierung kann man hier eine Datei auswählen.

Eine weitere Besonderheit ist das Klappmenü der Unterkategorie. Dieses wird über die Funktion:

```
update_sub_category
```

in dem Modul „CPFDatabase.py“ anhand des ausgewählten Eintrags in dem Klappmenü der Hauptkategorie aktualisiert. Dabei wird der aktive Eintrag gelesen. Danach wird mithilfe des Eintrages in der Konfigurationsdatei nach allen möglichen Unterkategorien gesucht. Diese List an Unterkategorien wird dann in dem Menü dargestellt. Es ist zu beachten, dass das Menü vor dieser Prozedur geleert wird, um Mehrfacheinträge bzw. falsche Einträge auszuschließen

Nachdem alle Einträge in dem Dialog einen Wert haben, kann man diesen Datensatz der Datenbank hinzufügen. Die Funktion hierfür heißt:

```
db_add_entry
```

in dem Modul „CPFDatabase.py“. Sie ließt alle Werte im Dialog aus, speichert sie in Variablen und fügt diese über die SQL-Funktion „INSERT“ in die Datenbank ein. Die ID in der Datenbank wird automatisch eingefügt.

Auch hier gibt es wieder zwei Besonderheiten. Die Erste ist, dass die Funktion versucht den Screenshot und das Symbol in ihren jeweiligen Ordner zu verschieben. Wenn dies schief läuft wird ein Fehler-Dialog angezeigt.

Wird einen Eintrag hinzu gefügt und ein Wert nicht Spezifiziert ist, wird auch ein Fehler-Dialog angezeigt.

3.6.5 Lesen der Einträge

Beim Lesen von Einträgen habe ich mich dafür entschieden mehrere Funktionen zu schreiben. Die erste ist:

`db_read`

die Zweite ist:

`read_attributes`

Die Funktion „db_read“ gibt alle Namen aller Programme wieder die eine spezifizierte Unterkategorie besitzen. Diese Funktion wird nur in der Anzeige der Unterkategorien und ihrer Programme benötigt. Dazu später mehr in der App-Anzeige.

Die Funktion „read_attributes“ hingegen ließt einen Gesamten Datensatz anhand seines Namens aus. Diese wird für alle übrigen Operationen genutzt. Der zurück gegebene Wert ist eine Liste. Der Index an dem sich eine Information befindet ist dabei gleich der Reihenfolge der Speicherplätze bei der Datenbankerstellung.

0. ID
1. Name
2. Kurze Beschreibung
3. Lange Beschreibung
4. Screenshot Ort
5. Ubuntu-Name
6. Debian-Name
7. Arch-Name
8. Symbol Ort
9. Haupt-Kategorie
10. Unter-Kategorie
11. Entwickler-URL

Achtung: Bei der Funktion „db_add_entry“ werden die Daten in einer Anderen Reihenfolge angesteuert. Die Reihenfolge innerhalb der Datenbank bleibt aber wie in der Liste beschrieben.

3.7 App-Anzeige

3.7.1 Anmerkung

Die App-Anzeige soll an der Wurzel die Hauptkategorien anzeigen. Gezeigt werden Icons und ihre Namen. Das Icon soll nativ aus dem System ausgelesen werden. Die Namen werden aus der Konfigurationsdatei ausgelesen.

3.7.2 Menü Ebenen

Wie schon erwähnt wird das Programm in der Hauptkategorie anfangen. Durch einen Klick auf die gewünschte Kategorie werden die Unterkategorien angezeigt.

Die Unterkategorien sind wie die Hauptkategorien in der Konfigurationsdatei festgelegt. Der Grund dafür ist das erstens die Übersetzung einfacher wird. Zum zweiten kann man selbst einfacher die Kategorien abändern wenn man eine eigene Datenbank aufbauen möchte. Der Nachteil ist, dass man zu einer Datenbank die richtige Konfigurationsdatei benötigt. Wenn man die falsche Konfigurationsdatei hat, werden nicht alle Kategorien bzw. die falschen angezeigt.

Wenn man die gewünschte Unterkategorie gewählt hat werden einem die Programme die mit dieser Kategorie und Unterkategorie in der Datenbank übereinstimmen angezeigt. Auch der Name und Pfad des Icons werden aus der Datenbank ausgelesen und als Icon des Programms verwendet. Ebenso der angezeigte Name.

Wenn man eines der angezeigten Programme anklickt öffnet sich der Programm-Dialog.

3.7.3 Inizialisierung

In dem Modul „Install_UI“ gibt es eine globale Variable mit dem Namen „stage“. Diese bekommt je nach derzeitig aktiver Kategorie einen String-Wert.

- Hauptkategorie = root
- Unterkategorie = sub
- Programmauswahl = Prog

Wenn das Programm gestartet wird ist der „stage“ wert auf „root. Es wird die Funktion

```
def set_to_start(builder)
```

ausgeführt. Wie man sieht hat sie als Argument wieder das Builder-Objekt aus dem Hauptprogramm. Danach wird eine weitere globale Variable eingeführt: „Plist“. Sie wird immer eine Liste der dargestellten Objekte halten.

Nun wird ein Listenobjekt erstellt welches immer das Icon und den Namen eines Objektes enthalten wird. Dieses Objekt kann in einem Gtk-Iconview als Icon mit Namen dargestellt werden.

Jetzt wird aus der Konfigurationsdatei Plist mit einer Liste der Hauptkategorien besetzt. Da man aus der Konfigurationsdatei nur einen String bekommt werden die Objekte an dem Zeichen „ , “ per split() getrennt und in die Liste eingefügt. Im Umkehrschluss müssen also alle Listenobjekte in der Konfigurationsdatei per Komma getrennt werden.

Danach wird in einer for-Schleife für jedes Objekt in der Liste ein Icon gesucht, als Pixbuf gespeichert und schließlich in die Liste des Iconview gespeichert. Wenn es kein Icon gibt wird das Standart-Fehler-Icon des Systems angezeigt.

Nachdem die Schleife fertig ist wird das Ergebnis dem Nutzer gezeigt.

Die Gesamte Funktion wird auch beim klick auf „Home“ wiederholt.

3.7.4 Von Hauptkategorie zu Unterkategorie

Wenn ein beliebiges Icon im Icon-View angeklickt wird, wird die Funktion

```
def Go_Down(builder,iconview,treepath)
```

ausgeführt. Ihre Argumente sind wieder der Builder, zu Interaktion mit dem Interface, der angeklickte Icon-View und sein Objekt-Baum „treepath“. Letzterer hält die für uns wichtige Information welcher Index in der Liste angeklickt wurde.

Die globale Variable „SelectedMain“ bekommt nun den Namen der gewählten Kategorie. Der Wert kommt aus der Plist-Variable. Wie schon erwähnt hält sie eine Liste der angezeigten Namen. Über „treepath“ finden wir den Index der angeklickt wurde und letztendlich darüber den Namen.

Nun wird der Icon-View geleert. Danach wird anhand der gewählten Kategorie aus der Konfigurationsdatei die Liste aller Unterkategorien geladen.

Genau wie bei der Initialisierung wird die fertige Liste erstellt (jetzt mit der globalen Variable „SubCategoryList“), Icons gesucht und das Ergebnis dem Benutzer gezeigt. Wichtig ist das die Variable „stage“ nun den Wert „Sub“ erhält da wir nun die Unterkategorien anzeigen.

3.7.5 Von Unterkategorie zu Programmauswahl

Wenn ein Icon angeklickt wird und „stage“ den Wert „sub“ hat wird die Funktion

```
go_Sub(builder,iconview,treepath,SelectedMain)
```

ausgeführt.

Über „treepath“ und die Globale Variable „SubCategoryList“ wird der Name der Ausgewählten Unterkategorie herausgefunden.

Nun kommt die schon beschriebene Funktion „db_read“ zum Einsatz. Sie ließt alle Programm der Unterkategorie aus und gibt eine Liste (Variable: „ProgramList“) mit ihnen zurück.

Für diese Liste wird jeweils im „Symbol“ Ordner der Datenbank nach dem Symbol des Eintrags gesucht. Das Symbol stellt nun das Icon des Objektes dar. Name und Programm-Icon werden als letztes wieder im Icon-View dargestellt.

Als letztes wird „stage“ auf „Prog“ festgelegt, da wir nun eine Auswahl an Programmen sehen.

3.7.6 Programm-Dialog

Wie schon erwähnt wird durch den Klick auf eines der angezeigten Programme der Programmdialog geöffnet.

Dabei wird die Funktion

```
def show_app(builder,iconview,treepath)
```

ausgeführt.

Anhand von „treepath“ und „ProgramList“ wird der Name des geöffneten Programm gefunden. Dann wird der Gesamte Datensatz über

```
read_attributes
```

in die Variable „Data“ gespeichert.

Die Werte werden dann an die vorgesehen Stellen in der Programm-Dialog Maske eingesetzt. Der Vorteil von diesem Vorgehen ist, dass ich nur einen Dialog-Maske erstellen muss, die in jedem Programm gleich angewandt wird. Ein weiterer Vorteil ist, dass der Benutzer immer das gleiche Layout sieht und sich so nicht immer wieder neu orientieren muss.

Wenn er das getan hat, wird die Variable „ProgramViewOpen“ auf „True“ gesetzt. Dadurch verhindert das Programm, dass die eine Maske mit anderem Inhalt noch einmal geladen wird. Man kann sich noch durch die Kategorien klicken. Wenn man allerdings versucht ein Programm-Dialog zu öffnen bekommt man einen Error-Dialog angezeigt.

3.7.7 Zurück Navigieren

Um Zurück zu navigieren wird die Funktion

```
def go_back(builder)
```

ausgeführt. Wenn „stage“ auf „root“ ist, muss das Programm nichts tun, da man nicht weiter zurück kann.

Wenn „stage“ auf „Sub“ ist, muss das Programm nur die Funktion „set_to_start“ ausführen um damit die Wurzel zu laden. Also die Ansicht der Hauptkategorien.

Wenn „stage“ auf „Prog“ steht wird es etwas komplexer. Dafür wird der Icon-View geleert. Danach werden aus der globalen Variable „SubCategoryList“ die Einträge der vorherigen Kategorie geladen. Genauso wie bei 3.7.4 werden nun die Icons geladen und dargestellt. Danach wird „stage“ auf „Sub“ gesetzt, da eine Unterkategorie angezeigt wird.

3.8 Installation

Abbildungsverzeichnis

4 Anmerkung zum Literaturverzeichnis

Das Literaturverzeichnis kennzeichnet Quellen mit englischen Inhalt mit „EN“.

Literatur

[1] EN: WebKit Hauptseite: https://webkit.org/

[2] EN: WebKit Lizens: https: https://webkit.org/licensing-webkit/ , gesamtes Dokument

[3] EN: Gtk Hauptseite: http://www.gtk.org/

[4] EN. LGPL Lizenz Webseite: http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

[5] EN: Kurzfassung des GTK: https://www.gimp.org/about/ancient_history.html

[6] EN: Beta-Launcher, Mein erstes GTK-Projekt: https://github.com/SiebenCorgie/Beta-Launcher

[7] EN: Unreal-Engine-4 Hauptseite: https://www.unrealengine.com/what-is-unreal-engine-4

[8] EN: Configparser Dokumentation: https://docs.python.org/3.5/library/configparser.html