GUI-Programmierung mit GTK+

Zweiter Teil

Florian Pelz

E-Mail: pelzflorian@pelzflorian.de

Heute...

- Distribution
- Mini-IMS
- Internationalisierung
- Weiterführendes

Letztes Mal

Wir haben behandelt

- wie man C-Programme schreibt,
- wie man in C mit GTK+ Fenster-Anwendungen programmiert,
- wie man den graphischen Glade-Editor dabei benutzen kann und
- ein kleines Programm entwickelt.

Probleme

Allerdings

- wollen Endnutzer ihre Software nicht erst kompilieren und
- das Kompilieren mit vielen Terminalbefehlen ist auch für Entwickler umständlich.

Maintainer

- Maintainer übernehmen eine wichtige Rolle für die Software-Infrastruktur.
- In einem Software-Projekt sorgen Maintainer dafür, dass die Software
  - leicht konfiguriert, kompiliert, getestet etc. werden kann,
  - auf allen Systemen läuft, wo sie laufen soll,
  - leicht an Endnutzer ausgeliefert werden kann und
  - die übrige Infrastruktur (Webseite, Wiki, Bug-Tracker, …) funktioniert,
  - gute und nur gute Patches akzeptiert werden,
  - der Change Log bzw. die News sauber geführt werden,
  - alle Autoren und Copyrights genannt werden,
  - …
- Maintainer für ein Betriebssystem nutzen die Infrastruktur, um die Software an Endnutzer auszuliefern.

Build Systems

- Ein *Build System* automatisiert das Kompilieren.
- Unabhängig von der Integrierten Entwicklungsumgebung.
- Notwendig für große Projekte.
- Bekannte Build Systems sind:
  - GNU Autotools
  - CMake
  - Waf, SCons
  - Ant, Gradle, Maven
  - *Meson*

GNU Autotools

- `autoconf` automatisiert
  - das Finden von Betriebssystemwerkzeugen,
  - das Feststellen von Betriebssystemfähigkeiten,
  - Compile-Time-Konfigurationen (z.B. Compiler-Auswahl).
- `automake` automatisiert
  - das Kompilieren der Software,
  - das Testen (Beweisen?) der Software,
  - das Installieren der Software und
  - den Release einer Version kompilierfähigen Quellcodes.
- Traditionell und üblich.
- Zum eigentlichen Bauen wird `make` verwendet.

Meson ools

- `meson` automatisiert
  - das Finden von Betriebssystemwerkzeugen,
  - das Feststellen von Betriebssystemfähigkeiten,
  - Compile-Time-Konfigurationen (z.B. Compiler-Auswahl).
- `meson` automatisiert
  - das Kompilieren der Software,
  - das Testen (Beweisen?) der Software,
  - das Installieren der Software und
  - den Release einer Version kompilierfähigen Quellcodes.
- Traditionell und üblich.
- Zum eigentlichen Bauen wird `ninja` endet.

Meson

- `meson` ist ein *einfach benutzbares* Build System.
- Meson ist deklarativ und *nicht* Turing-vollständig.
- Viele GNOME-Projekte steigen von Autotools auf Meson um.

- Einführung $\longrightarrow$ Handout.
- Genauere Anleitungen auf `http://mesonbuild.com`.

Versionskontrollsysteme

- Versionskontrollsysteme wie `git` verwalten mehrere Versionen des Quellcodes.
- Sie erlauben
  - paralleles Arbeiten mehrerer Entwickler am gleichen Code,
  - paralleles Arbeiten eines Entwicklers an unterschiedlichen Teilen des Codes,
  - eine leichte Rückkehr zu alten Versionen und
  - leichtes Finden der Version, mit der ein Bug zum ersten Mal auftrat.

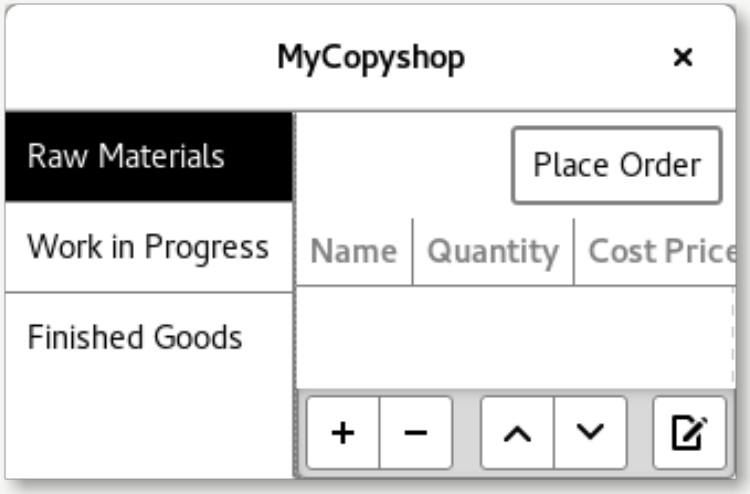
Für die Endnutzer: Pakete

- Auf GNU-Systemen wird Software oft mit Paketmanagern ausgeliefert.
- Pakete bestehen meist hauptsächlich aus
  - entweder getesteten automatischen Skripts zum Kompilieren und Installieren
  - oder fertigen Dateien der Software, die durch ein Skript an ihren betriebssystemspezifischen Platz gebracht wurden.
- In der Regel heißt das, Pakete werden durch Aufruf eines Build Systems erstellt.
- Details hängen vom Paketmanager ab.

Für die Endnutzer: Bundles

- Auf den meisten Betriebssystemen sind auch (evtl. erst nach Installation) selbstständig lauffähige Bundles üblich.
- Ein Bundle ist ein Verzeichnis mit der Software *und allen Abhängigkeiten*.
- Metadaten und Konfigurationsdateien befinden sich unter Umständen außerhalb des Bundles.
- Ein solches Bundle kann dann z.B. zum Download im Web, in einem Appstore oder auf physischen Datenträgern vertrieben werden.
- Sicherheitsrisiko? ($\longrightarrow$ Flatpak)
- Beizufügen ist ein Angebot, den GTK+-Quellcode gegen angemessen niedrige Gebühr zu liefern, „complete and corresponding“ (geänderter Quellcode muss nutzbar sein).

Mini-IMS



- Wir bauen uns zum Üben ein kleines Inventarverwaltungssystem für einen fiktiven Copyshop.
- Da das ein „großes Projekt“ ist, verwenden wir Objekte:
  - Vererbung.
  - Modularisierung.
  - Andere Sprachen.

Lokalisierung

- Bezeichnet das Anpassen einer Software an die lokale Sprache und Kultur.
  - Übersetzung,
  - Darstellung von Datum und Uhrzeit,
  - Maßeinheiten,
  - Dezimaltrennzeichen,
  - ...
- Kurz: l10n.

Internationalisierung

- Bezeichnet das Bereitstellen der zur Lokalisierung nötigen Infrastruktur.
- Kurz: i18n.
- PO-Dateien (wie „`de.po`“) beinhalten Übersetzungen zu allen sichtbaren Zeichenketten.
- Im Code benutzt man statt

  ```
  label = gtk_label_new ("Hello␣World");
  ```

  nun

  ```
  label = gtk_label_new (_("Hello␣World"));
  ```

- Die Funktion `_()` von GNU Gettext liefert die passende Übersetzung.
- Das `_()` kann auch als „Markierung“ der Zeichenkette verstanden werden.
- Ähnliche Markierungen geben noch Kontext für Übersetzer mit, z.B. wo im Programm die Zeichenkette auftaucht.
- Siehe Handouts.

Accessibility

- D.h. Software Unterstützungsprogrammen zugänglich machen und „behindertengerecht“ gestalten.
- Accessibility umfasst:
  - Kompatibilität mit Screenreadern,
  - Kompatibilität mit großer Schrift,
  - Farbwahl und Kontrast,
  - einstellbare Doppelklickdauer,
  - Aufleuchten des Bildschirms bei Fehlern,
  - ...
- Kurz: a11y.
- In GTK+ größtenteils automatisch.
- Aber ATK sollte z.B. zum Angeben von Beziehungen zwischen Widgets benutzt werden.

Unit-Tests

- Fehler im Programm findet man u.A. durch Testen.
- GLib enthält ein Framework für Unit-Tests.
- Meson kann darin geschriebene Tests ausführen.
- Siehe z.B. den Quellcode von GTK+ für eine Beispiel-Testsuite.
- reftests vergleichen das Aussehen Pixel für Pixel
  $\rightarrow$ wird in GTK+ benutzt.

Statische und dynamische Analyse

- Statisch: Ohne das Programm laufen zu lassen.
  - Mehrere Compiler.
  - coala für Coding-Style-Fehler.
  - ...
- Dynamisch:
  - Valgrind.
  - Profiler.

Profiling

- Ein Profiler misst, welche Teile des Programms
  - wie oft laufen und
  - wie lange brauchen
  - und/oder sonstige Ressourcennutzung (z.B. Speicher).
- So weiß man, welcher Code optimiert werden sollte.

gtk-doc

- Code kann in den Quelldateien dokumentiert werden.
- gtk-doc erzeugt daraus eine HTML-Dokumentation.
- Diese kann z.B. mit Devhelp gelesen werden.
- Aber  – GNOME Builder benutzt reStructuredText/Sphinx, Builders größter autor Christian Hergert ist aber auch damit nicht völlig glücklich.

Introspection

- Erlaubt das Abfragen von Informationen über Typen und Klassen:
  - Methodennamen,
  - Signale,
  - Properties,
  - ...
- Erlaubt automatisches Generieren von Bindings für andere Programmiersprachen.

D-Bus

- Für Inter-Process Communication.
- Insbesondere für Kommunikation mit der Benutzeroberfläche:
  - Aufrufen von Aktionen im Programm,
  - Notifications,
  - ...

CSS

- CSS ermöglicht es,
  - das Aussehen von Widgets und auch
  - konfigurierbare Tastaturbelegungen

  in GTK+ zu verändern.

Weiteres

- GSettings speichert Einstellungen, alten Programmzustand etc.
- GResource ermöglicht das Kompilieren von Binärdaten in die Anwendung hinein.
- GtkInspector zeigt und ändert GTK+-Informationen zu laufenden Programmen.
- `g_autoptr` lässt Speicher automatisch freigeben.
- Human Interface Guidelines beachten.
- Nicht alles braucht eine GUI.

Weiterführend

- GLib- und GTK+-Lehrbuch:
  `https://people.gnome.org/~swilmet/glib-gtk-dev-platform.pdf`
- Für C++-Programmierer:
  `https://developer.gnome.org/gtkmm-tutorial/stable/`
- Internationalisierung:
  `https://wiki.gnome.org/TranslationProject/DevGuidelines`

Bildquellen etc.

- Folien-Design basiert auf `https://git.gnome.org/browse/presentation-templates/`.
- Baustellen-Warnschild von `https://openclipart.org/detail/3850/work` quasi gemeinfrei (CC0) von dchandir.