Gimp

Marc Alexander Lehmann

1999-02-15

Zusammenfassung
Vortrag und „So wird's gemacht“ auf dem 1. deutschen Perl-Workshop

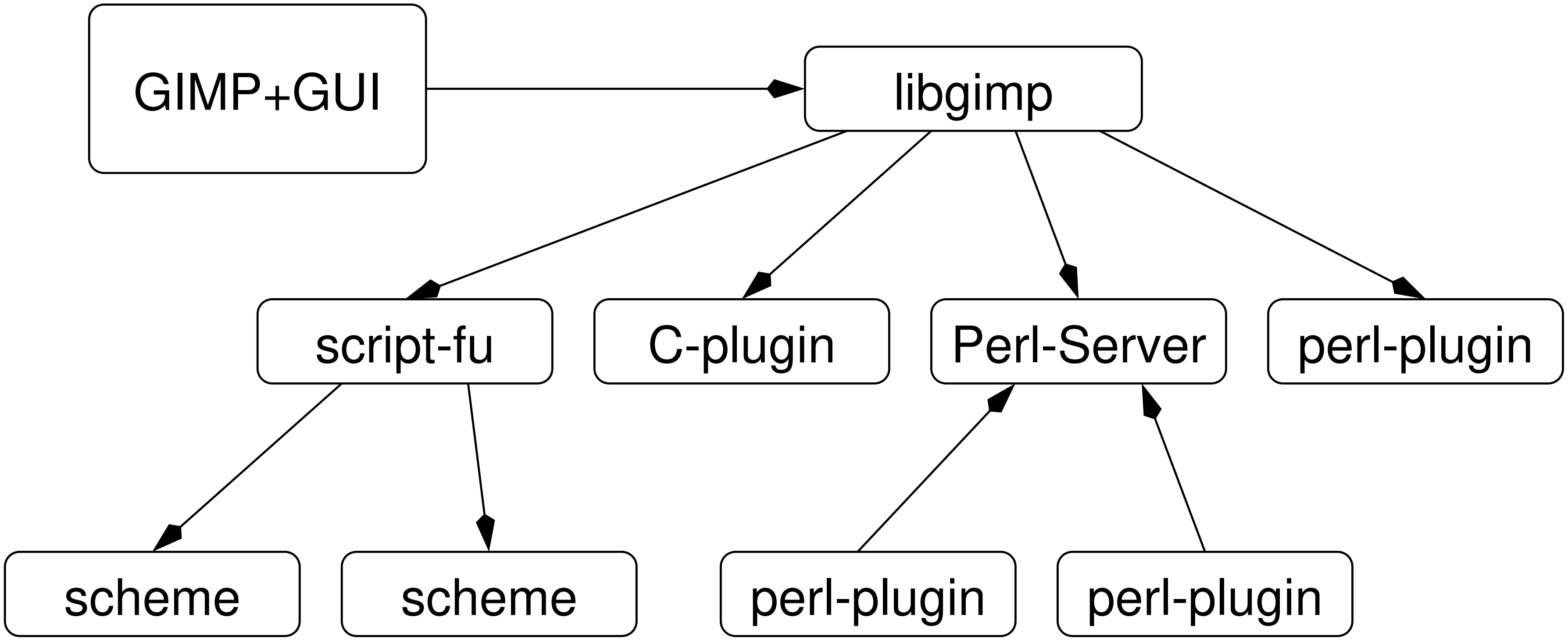
Teil 1 — Gimp programmieren

Gimp – das Programm

- 'C'-Programme

  GIMP **G**NU **I**mage **M**anipulation **P**rogramm, basierend auf:
  GTK+ **G**imp/**G**NU Tool**k**it
  GLIB die **G**NU **Lib**rary, eine Art verbesserte libiberty (`-liberty`)

Überblick über das System



Das Plug-in: C-Stil

```
use Gimp;

# Initialisierung
...
exit main;

sub query {
   Gimp->install_procedure "Funktionsname", ...
}

sub <callback> {...}

sub net {...}
```

Nachteile

- Umständliche Unterscheidung zwischen „Kommandozeile“ (net) und „Plug-in“ (`query`)
- Standardargumente (z.B. `run_mode`) müssen manuell behandelt werden

Vorteile

- erlaubt kurze „nur Kommandozeile“-Plug-ins:

```
use Gimp;

sub net {
   ... # auf geht's!
}

exit main;
```

Das Plug-in: `Gimp::Fu`-Stil

```
use Gimp;
use Gimp::Fu;

register <viele, viele Argumente>,
         <coderef>

exit main;
```

Nachteile

- `register` erwartet sehr viele Parameter (am besten kopiert man ein vorhandenes Plug-in)

Vorteile

- keine Unterscheidung zwischen Aufrufen von der Kommandozeile oder aus dem GIMP heraus
- automatisch netzwerkfähig
- einheitliches Aussehen *und* Auftreten der Skripte
- viele *convinience*-Features:
  - automatisches Auswerten von Standardargumenten (`RUN_WITH_LAST_VALS` etc.)
  - konfigurierbares Gtk-Dialogfenster bei interaktivem Aufruf
  - Parsen von Rückgabewerten (z.B. Anzeigen oder Abspeichern des Ergebnisbildes)

Beispielplugin

```
use Gimp;
use Gimp::Fu;

register "drawable_rotate", "rotate an image",
         "this plug-in ...", "Marc Lehmann", "Marc Lehmann",
         "1999-02-14", "<Image>/Layers/Rotate",
         "*",
         [[PF_FLOAT,   "angle", "the rotation angle"]],
   sub {
      my ($image, $drawable, $angle) = @_;
      $drawable->rotate (1, $angle);
      $image;
   }

exit main;
```

Der DB-Browser

(Vorführung)

entsprechender Eintrag in der PDB:

```
"perl_fu_rotate_diagonal"
"rotate an image diagonally" "this plug-in...."
"Marc Lehmann" "Marc Lehmann" "1999-02-14"
"GIMP Plug-in"

Argumente:

run_mode   INT32     "Interactive, [non-interactive]"
image      IMAGE     "the image"
drawable   DRAWABLE  "the drawable"
angle      FLOAT     "the rotation angle"
```

entsprechende Kommandozeilenargumente:

```
Usage: ./beispiel.pl [gimp-args..] [interface-args..] [script-args..]
   gimp-arguments are
      -gimp <anything>          used internally only
      -h | -help | --help | -? print some help
      -v | --verbose            be more verbose in what you do
      --host|--tcp HOST[:PORT] connect to HOST (optionally using PORT)
                                (for more info, see Gimp::Net(3))
   interface-arguments are
      -o | --output <filespec> write image to disk, then delete
      -i | --interact           let the user edit the values first
   script-arguments are
      -angle float              the rotation angle
```

Gimp-Datentypen

- `PF_INT8, PF_INT16, PF_FLOAT, PF_STRING, PF_VALUE`
- `PF_COLOUR, PF_COLOR, [123,50,70], "papayawhip", "#4500ff"`
- `PF_IMAGE` ganze Bilder
- `PF_LAYER` Ebenen, Schichten, Masken
- `PF_CHANNEL`, Farbkanäle, Auswahlen
- `PF_DRAWABLE == PF_LAYER | PF_CHANNEL | PF_SELECTION`

Gimp::Fu-Datentypen

- PF_FONT
- PF_BOOL, PF_TOGGLE
- PF_SLIDER
- PF_SPINNER
- PF_BRUSH
- PF_PATTERN
- PF_GRADIENT

Syntax: prozedural. . .

- Gimp enthält eine **P**rozedurale **D**aten**b**ank (PDB)
- darin sind alle Funktionen (intern, plug-ins) mit Argumenten und Rückgabewerten beschrieben
- enthält lediglich Prozeduren bzw. Funktionen, aber *keine* Methoden

- Funktionen sind manchmal nach Funktionsgruppen geordnet:

```
gimp_drawable_fill
gimp_drawable_get_pixel
gimp_drawable_height
gimp_drawable_width
...
```

- Manchmal aber auch nicht:

```
gimp_attach_parasite
gimp_find_parasite
gimp_parasite_list
parasite_new
```

- Alle Funktionen können mit dem `:auto`-Tag (dem default!) importiert werden:

```
use Gimp;
...
$height = gimp_drawable_height ($image, $layer);
gimp_drawable_fill ($layer, WHITE_IMAGE_FILL);
```

- oder, als Methode, über das Gimp-Modul:

```
use Gimp ();
...
$height = Gimp->drawable_height ($image, $layer);
```

. . . vs. objektorientiert

- Die prozedurale Syntax führt zu schwer lesbaren Skripten, sowohl in Script-Fu als auch in Perl.
- Das Gimp-Modul kennt bietet Methoden, mit der sich die Syntax vereinfachen und abkürzen läßt.
- Grundidee: die meisten Funktionen, die mit `DRAWABLE`s arbeiten, haben Namen, die mit „`gimp_drawable_`“ beginnen, für andere Objekte gilt das gleiche.

- Einführung von Pseudoklassen, die einen „Suchpfad“ über den Namensraum enthalten, z.B.:

Klassen	Suchpfad
`Gimp::Image`, `Image`	`gimp_image_`, `gimp_drawable_`, `gimp_`, `plug_in_`
`Gimp::Palette`, `Palette`	`gimp_palette_`
`Gimp::Channel`, `Channel`	`gimp_channel_`, `gimp_drawable_`, `gimp_selection_` . . .

✘ Beispiel: die Aufrufe

```
Channel->height ($channel);
$channel->height;
```

versuchen zuerst `gimp_channel_height` (existiert nicht), dann `gimp_drawable_height` (existiert) aufzurufen.

- Vernünftige Benutzung

```
$image = new Image 300,200,RGB;
       = gimp_image_new (300,200,RGB);
```

- Mißbrauch ist jederzeit möglich:

```
Channel->gimp_image_new (300,200,RGB);
```

- Einige (redundante) Argumente können auch einfach weggelassen werden:

  ✘ Beispiel: reduntantes `Image`-Argument („was wird invertiert"?):

  ```
  gimp_invert ($image, $layer);
  # ist identisch zu
  gimp_invert ($layer);
  ```

  – Vorteil: Zukunftssicher

  ✘ Beispiel: Standardwert für `run_mode` (erlaubt OO-Syntax):

  ```
  plug_in_vinvert (RUN_NONINTERACTIVE, $image, $drawable);
  # ist identisch zu
  $drawable->vinvert;
  ```

Debugging

- Tracing:

✘ `$drawable->height;`

```
Gimp::set_trace(TRACE_CALL)
  gimp_drawable_height(2) = (512)
Gimp::set_trace(TRACE_NAME)
  gimp_drawable_height(drawable=2) = (height=512)
Gimp::set_trace(TRACE_ALL)
  gimp_drawable_height(
          DRAWABLE drawable=2 "the drawable"
          ) = (
          INT32 height=512 "height of drawable"
          )
```

- ✘ `Gimp::set_trace(*STDERR);`
- ✘ `Gimp::set_trace(\$trace_string);`

- (Meine) übliche Vorgehensweise beim Debuggen:
  1. GIMP starten
  2. Perl-Server starten
  3. test-skript starten, verbessern, starten, . . .

Probleme & Lösungen

- gleichzeitiger Zugriff mehrerer Clients
  - locking
  - private GIMP instance
- `TRUE` $\Rightarrow$ `"TRUE"` $\Rightarrow$ 0 $\Rightarrow$ „false“

Teil 2 — Gimp, das Design

- Motivation: die Entstehungsgeschichte
- welche Features halfen bei der Umsetzung?
- Implementationsdetails

Entstehung

Motivation: Script-Fu $\Longleftrightarrow$ 'Scheme' $\Longleftrightarrow$ *Frust*

1998-02-06 T 17:51:56 +01:00 0.01 h2xs-Version

1998-02-07 T 04:18:17 +01:00 0.02 funktioniert!

1998-02-07 T 23:31:08 +01:00 0.05 erste Version auf CPAN

Feature #1: Perl-XS

- XS ist speziell auf die Anforderungen von Perl $\Leftrightarrow$ C Schnittstellen zugeschnitten
- h2xs (schnelle Erzeugung von Modul-Prototypen)

Feature #2: Gimp-PDB

- Gtk: tolles design, nicht skriptsprachenfreundlich
- Gimp: (ehemals) ad-hoc-design, sehr skriptsprachenfreundlich

Feature #3: CPAN und die Gimp Plug-in Registry

Beide „registries“ sind

- übersichtlich
- benutzerfreundlich (automatische updates)
- betreut (Ansprechpartner sind erreichbar und helfen gerne)
- bekannt bei der jeweiligen Benutzergruppe

Feature #4: Module

Gtk das Perl-Interface zu GTK+

PDL die Perl Data Language („piddle“), effizientes manipulieren von Pixeldaten

Data::Dumper zum Serialisieren von Daten

Storable noch besser zum Serialisieren von Daten geeignet

- wird noch nicht verwendet, weil zusätzliche Abhängigkeit

Parse::RecDescent zum Parsen von Script-Fu-Skripten

Feature #5: TIMTOWTDI

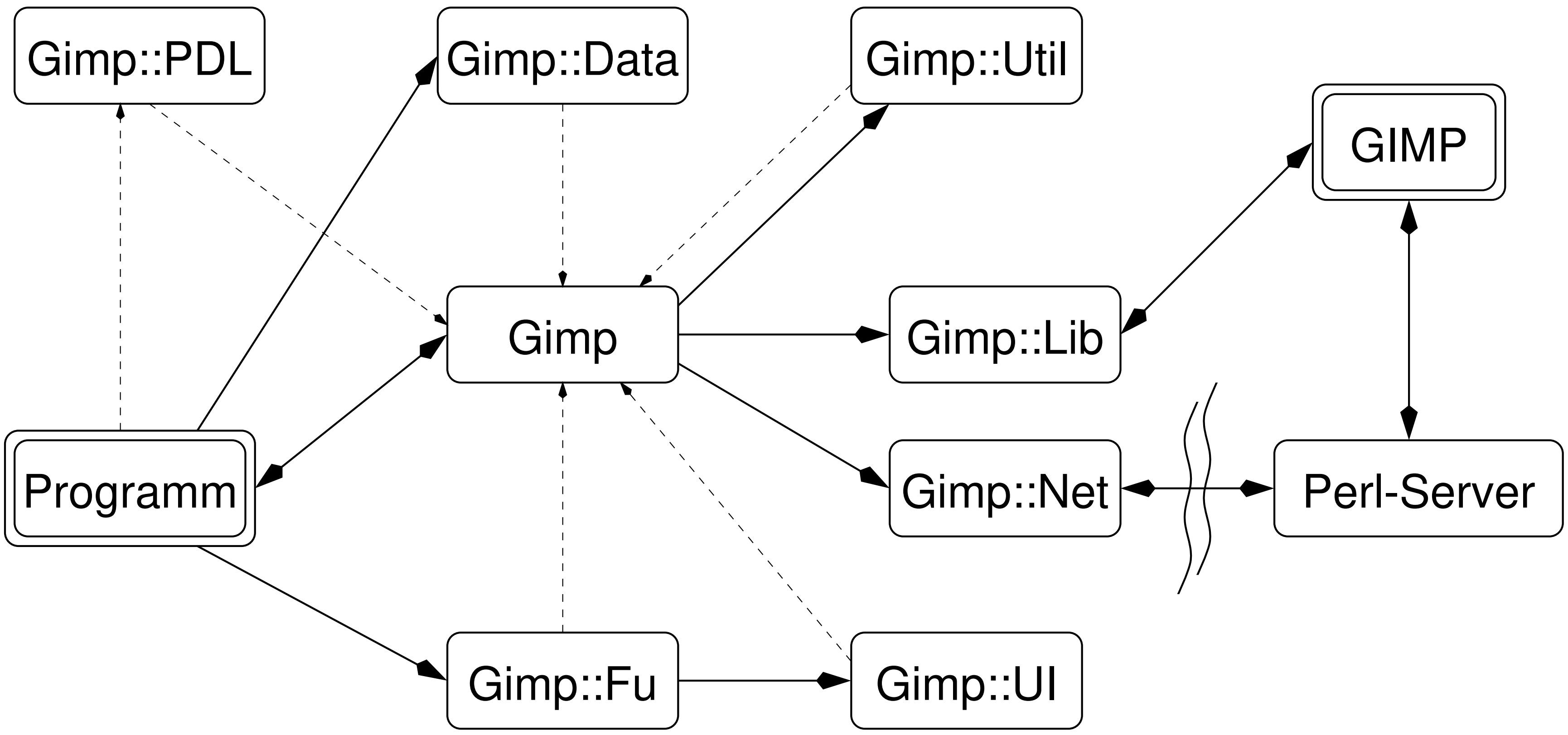
- Gimp-API: prozedural und objekt-orientiert
- Aufrufe
  - Kommandozeile
  - Netzwerk
  - als „ganz normales“ Plug-in

Feature #6: Perl

Die dynamische Natur von Perl erlaubt es:

- beliebige Namensräume schaffen (`AUTOLOAD` & `gimp_call_procedure`)
- Funktionen dynamisch überschreiben (`Gimp::PDL`)
- Funktionen abzufangen (`Gimp::Net`)
- debugging, Ablaufverfolgung
- Exception-Handling (Fehlerbehandlung & Weiterleitung)

Die Gimp-Module



Die AUTOLOAD-Funktion(en)

Jeder Gimp-Funktionsaufruf geht beim ersten Mal über die `Gimp::AUTOLOAD`-Funktion. Sie prüft:

- soll Funktion ignoriert werden? `exists $ignore_function{$sub}`
- existiert Funktion im Interfacemodul? `UNIVERSAL::can($interface, $sub)`
- existiert Funktion in der PDB? `_gimp_procedure_available ($sub)`
- existiert Funktion in `Gimp::Util`? `UNIVERSAL::can(Gimp::Util, $sub)`
- nicht gefunden? dann nächstes Präfix ausprobieren.

Die AUTOLOAD-Funktion erzeugt dann dynamisch einen „wrapper“, der auf die Klasse und die Methode zugeschnitten ist.

Die Schnittstellen-Schnittstelle

Ein Schnittstellen-Modul muß lediglich die folgenden Funktionen zur Verfügung stellen:

- `gimp_main`, Initialisierung
- `_gimp_procedure_available ($funktionsname)`, um zu prüfen, ob eine Funktion existiert
- `gimp_call_procedure ($funktionsname, argumente...)`, um eine Funktion aufzurufen
- `set_trace (...)`, um die Ablaufverfolgung zu steuern

`Gimp::Lib`

- enthält im wesentlichen die benötigten Schnittstellenfunktionen, die in XS implementiert sind

- einige Funktionen (z.B. `gimp_list_images`) werden überschrieben (Typsicherheit)

`Gimp::Net`

- Das GIMP ist *keine* Funktionsbibliothek, ein Proxy-Server wird benötigt
- Proxy-Server ist ein spezielles Plug-in, der `Perl-Server`.
- der Proxy-Server kann manuell (Menü) gestartet werden oder per Kommandozeilenargument gestartet werden
- Verbindungen über TCP/IP, Unix Domain Sockets (und socketpair!).
- optional wird ein eigener GIMP-Prozess gestartet (eine private Instanz), mit eigenem Perl-Server
- Passwortschutz (gleiche Sicherheit wie telnet)

Das `Gimp::Net`-Protokoll

Kommunikation geschieht über ein streamfähiges Protokoll, die Perl-Funktionen `Gimp::Net::args2net` und `Gimp::Net::net2args` serialisieren und deserialisieren die Argumentliste.

- `args2net` erzeugt einen String, der gültigen Perl-Code darstellt, dessen Ergebnis das Argumentarray darstellt.
- `net2args` besteht im wesentlichen aus einem `eval "($_[0])"`, das den Perl-Code ausführt (ohne Überprüfung)
- geringfügige Kompression, `bless \(my $x=$object),$package` wird beispielsweise als `b($object,$package)` dargestellt

- `Storable` wäre sicherer, schneller und kürzer (und eine Modulabhängigkeit mehr. . .)

- `_gimp_procedure_available` ist z.B. sehr kurz:

```
my $req="TEST".$_[0];
print $server_fh pack("N",length($req)).$req;
$server_fh->read($req,1);
return $req;
```

- Fehlerbehandlung:

  Im Server:

  `@args=deobjectify(eval { Gimp->$req(objectify(@args)) });`

  (der Aufruf wird durch ein `eval {}` geschützt, der Fehlerstring wird Teil des Antwortpacketes)

  Im Client wird `croak $error` aufgerufen, der Server bleibt trotz grober Fehler (z.B. falsche Syntax) einsatzfähig.

- Komplexe Objekte:

  – die meisten Objekte sind lediglich mit `bless` behandelte Referenzen auf Integerzahlen (libgimp-Protokoll), sie werden „transparent“ durchgereicht.
  – manche Objekte (z.B. PixelDaten) haben assoziierte C-Strukturen, die nicht über das Netzwerk transportiert werden, *deshalb*:

- Server-seitiger Cache:

```
if($object_dynamic{ref $_}) {
   $objects{++$object_uid}=$_;
   $_=bless \(my $x=$object_uid),ref $_;
}
```

- und DESTROY-Funktionen auf Client-Seite:

```
*Gimp::Tile::DESTROY =
*Gimp::PixelRgn::DESTROY =
*Gimp::GDrawable::DESTROY = sub {
   my $req="DTRY".args2net(@_);
   print $server_fh pack("N",length($req)).$req;
};
```

ENDE

Literatur

[1] *Die Gimp-Perl-Seite*: `http://www.goof.com/pcg/marc/gimp.html`

[2] *Das GIMP*: `http://www.gimp.org/`

[3] *Programmieren von Grafiken mit Perl*, Marc Lehmann, O'Reilly-Verlag 1999

[4] *Haskell*: `http://www.haskell.org/`