MP3s kategorisieren und Playlisten anlegen

Trainierter DJ

Je nach Stimmungslage haben Musikliebhaber mal Lust auf Rock, mal auf Schmusepop. Ein MP3-Player mit grafischer GTK-Oberfläche wählt aus der privaten Sammlung passende Lieder aus, erstellt eine Playliste und spielt sie ab. Das Perl Object Environment sorgt dafür, dass alles flüssig abläuft. Michael Schilli

www.photocase.de

In jeder größeren MP3-Sammlung aus gerippten CDs finden sich immer wieder ungeahnte Schätze: Eine Computer-gesteuerte Zufallsauswahl fördert aus Tausenden von MP3-Dateien erstaunliche Kostbarkeiten zutage und erlaubt überraschende Zusammenstellungen aufgrund einfacher Kriterien.

Das hier vorgestellte Perl-Skript »rateplay« zeigt dem Benutzer alle Musikstücke in zufälliger Reihenfolge. Er bewertet dann jedes einzelne nach zwei Kriterien: dem so genannten Energize- und dem Schmoop-Faktor. Energize bestimmt die Dynamik des Stücks, Schmoop die Schmusigkeit. Auf einer Skala von eins bis fünf hätte etwa der Song „Thunderstruck“ von AC/DC einen Energize-Faktor von fünf und einen Schmoop-Faktor von eins. „Don’t Know Why“ von Norah Jones hätte einen Energize-Faktor von eins und einen Schmoop-Faktor von fünf.

Jedes Mal, wenn der Benutzer seine Stimme abgibt, speichert eine Datenbank den Dateisystempfad zu dem Song und die beiden Faktoren. Haben sich so einige Bewertungen angesammelt, erstellt das Skript auf Anfragen wie „Spiele mir schnelle Songs, aber keine, die meine Freundin verscheuchen“ Playlisten und spielt sie ab.

Grafisches Interface

Abbildung 1 zeigt das Skript in Aktion. Zum Abspielen bereits bewerteter Songs wählt der Musikfreund die gewünschten Energize- und Schmoop-Faktoren in den oberen zwei Knopfreihen aus. Ein Mausklick auf »Play Rated« erstellt eine Playliste aus jenen Songs der Datenbank, zu denen die Werte passen, und spielt sie der Reihe nach ab. Über »Play Next« und »Play Previous« springt das Programm zum nächsten und vorigen Stück.

Um neue Lieder zu bewerten, klickt der Benutzer auf den Button »Random Rate«. Rateplay erstellt daraufhin eine Playliste aus bisher unbewerteten Songs und spielt sie ab. Währenddessen lassen sich für jedes Stück die einzelnen Level einstellen, und zwar über die Buttonleiste am unteren Rand. Nur jeweils ein Energize- und Schmoop-Level pro Stück ist erlaubt. Ein Klick auf »Rate« speichert die Werte in der Datenbank und Rateplay springt zur nächsten Datei.

Vom Duo zum Trio

Rateplay nutzt gleich mehrere Perl-Module: Zu dem aus [2] bekannten Erfolgsduo POE und GTK für ruckfreie GUIs gesellt sich diesmal noch der Kommandozeilen-MP3-Spieler Musicus [3] von Robert Muth dazu, ein C++-Programm, das auf den dynamischen Libraries des bekannten Players Xmms aufsetzt. Das Modul »POE::Component::Player::Musicus« (im Folgenden durch »PoCo::Player::Musicus« abgekürzt) von Curtis Hawthorne bindet den MP3-Spieler in den POE-Reigen ein – so steuert das GUI den Player ruckelfrei.

Zum Speichern der Bewertungen nutzt Rateplay die in [4] vorgestellte objektorientierte »Class::DBI«-Abstraktion, die diesmal eine SQLite-Datenbank nutzt (siehe [5]). SQLite legt in einer einzigen Datei eine professionelle Datenbank mit SQL-Abfragemöglichkeit ab. Natürlich gibt es dafür auch ein Perl-Modul aus der DBI-Reihe auf dem CPAN. SQLite ist im Grunde eine ganz normale SQL-Datenbank. Um festzustellen wie viele bewertete Songs in der definierten Tabelle »rated_songs« stehen, dockt der Anwender einfach mit dem Kommandozeilen-

Tool »sqlite« an die von Rateplay erzeugte Datenbankdatei »rp.dat« an und setzt einen SQL-Befehl ab:

```
$ sqlite rp.dat
SQLite version 2.8.12
Enter ".help" for instructions
sqlite> select count(*) from rated_songs;
887
```

In diesem Beispiel sind also 887 Songs schon bewertet. Genug Stücke, um erstaunliche Playlisten zu erstellen!

Rateplay im Detail

Das Programm Rateplay ist sehr umfangreich. Listing 1 zeigt den Source-Code, den es auf [1] zum Download gibt. Die Konfigurationszeilen acht bis zehn legen den Pfad zur Datenbankdatei fest (in der Variablen »$DB_NAME«) und das Verzeichnis (in »$SONG_DIR«), in dem das nachstehend definierte »find«-Programm rekursiv nach Dateien mit der Endung ».mp3« suchen soll.

Die globalen Arrays »@PLAY_ENERG« und »@PLAY_SCHMO« speichern die Werte der Checkbuttons für die Songauswahl am oberen Rand des GUI. Die Skalare »$RATE_ENERG« und »$RATE_SCHMO« hingegen enthalten die Einstellungen der Radiobuttons am unteren Rand des Fensters und nehmen Werte von eins bis fünf für Energize- und Schmoop-Faktor entgegen. Die beiden Arrays »@RATE_ENERG_BUTTONS« sowie »@RATE_SCHMO_BUTTONS« führen die Objekte der Radiobuttons als Elemente, damit das GUI die für einen Song in der Datenbank gefundenen Bewertungen gleich voreinstellt.

Die Klasse »Rateplay::DBI« ab Zeile 27 erbt von »Class::DBI« und definiert die objektorientierte Abstraktion auf die SQLite-Datenbank. Falls diese noch nicht existiert (was bei SQLite daran zu sehen ist, dass es die entsprechende Datei nicht gibt), legt der SQL-Code ab Zeile 37 die Datenbankdatei samt benötigter »rated_songs«-Tabelle mit den Spalten »path« (Pfad zur MP3-Datei), »energize« (für den Energize-Level) und »schmoop« (Schmoop-Level) an. Zeile 43 führt die Arbeit aus.

Das in Zeile 30 hereingezogene »Class::DBI::AbstractSearch« erlaubt später über die Funktionalität von »Class::DBI« hinausgehende And-/Or-Abfragen mit »Rateplay::Song->search_where()«. Die OO-Abstraktion auf die Tabelle legt die Klasse »Rateplay::Song« ab Zeile 47 an. Damit ist der Rest des Skripts frei von SQL-Statements.

MP3-Player mit POE ansteuern

Das Hauptprogramm steht im Paket »main« ab Zeile 56. Es definiert die POE-Session, die das GUI und den Player betreibt. Das mit dem Parameter »package_states« referenzierte Array legt eine Liste von später im Skript definierten Funktionen fest, die von gleichnamigen POE-Events aufgerufen werden. Ruft das Hauptprogramm beispielsweise die »getpos()«-Methode des Players auf, antwortet dieser mit der Position im aktuell gespielten Song, indem er der POE-Hauptsession einen »getpos«-Event schickt. Dank der obigen »package_states«-Deklaration weiß die Session nun, dass sie in diesem Fall die ab Zeile 79 definierte Funktion »getpos()« anspringen muss. Den Ablauf der kompletten Session zeigt Abbildung 2.

Ähnlich verhält es sich mit »getinfocurr«: Gemäß der Dokumentation von »PoCo::Player::Musicus« löst der Player diesen Event aus, wenn das Skript die »getinfocurr()«-Methode des Player-Objekts aufruft. Die Callback-Funktion erhält als Parameter den Interpreten, Titel und einige weitere MP3-Informationen des gerade abgespielten Stücks. Die Zeilen 92 und 93 frischen die Interpret- und Titelanzeige im GUI auf.

Den »song«-Event löst das Skript selbst aus. Immer wenn der Player ein neues Stück spielen soll, schickt Rateplay – wie in Zeile 307 – einen »song«-Event an die in Zeile 143 getaufte Session »main«, also an die POE-Hauptsession. Diese ruft daraufhin die ab Zeile 97 definierte gleichnamige »song()«-Funktion auf, die den Pfad der MP3-Datei als erstes Argument aufschnappt, den Player stoppt und ihn dann die neue MP3-Datei abspielen lässt.

Der »scan_mp3s«-Event wird in Zeile 75 kurz nach dem Systemstart ausgelöst und lässt das Skript in die ab der Zeile 107 definierte Funktion »scan_mp3s« springen. Diese holt mittels »retrieve_all()« alle bewerteten Songs aus der Datenbank und legt sie als Schlüssel im globalen Hash »%RATED« ab. Dann startet sie einen Kindprozess in einer »PoCo::Child«-Session, der das externe »find«-Kommando aufruft und MP3-Dateien auf der Festplatte aufstöbert. Wenn Find eine Datei gefunden hat, schreibt es den Pfad immer nach Stdout.

Die Session springt dann aufgrund der Event-Definition in Zeile 113 (und der »package_states«-Definition in Zeile 64) in die »mp3_stdout()«-Funktion, die ab Zeile 328 definiert ist. Sie hängt den Dateinamen an das globale »@MP3S«-Array an, falls der Anwender das Stück noch nicht bewertet hat. Zeile 336 frischt die Statusanzeige über die ablau-

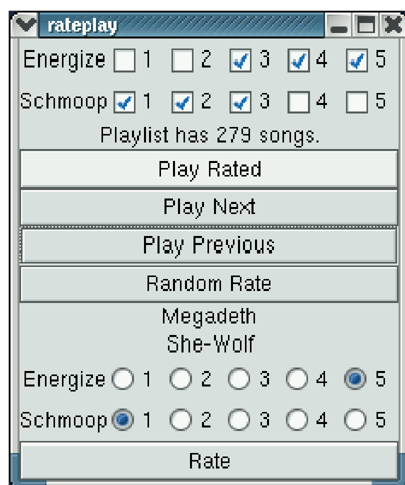

Abbildung 1: Hier spielt Rateplay nur Songs mit den Energize-Werten drei bis fünf und Schmoop eins bis drei ab. Im Moment dröhnt ein Heavy-Metal-Song aus den Lautsprechern, mit einem Energize-Level von fünf und einem Schmoop-Wert von eins.

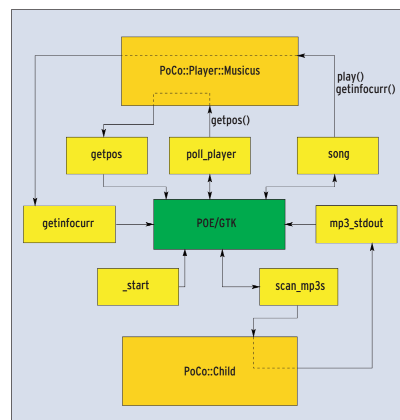

Abbildung 2: Mit POE springt das Programm zwischen verschiedenen Zuständen hin und her. Die »PoCo«-Prozesse laufen parallel. Doppelpfeile signalisieren Zustände, die der POE-Kernel kurz anspringt und dann parallel weiterlaufen lässt.

fende Suche auf. Wie in [2] nachzulesen ist, arbeitet POE mit ungewöhnlichen Parameter-Konstanten. »ARG0« ist zum Beispiel der Index, an dem der Parameter in »@_« steht, der dem Event übergeben wurde. Der von »PoCo::Child« gesetzte erste Parameter ist eine Referenz auf einen Hash, der unter dem Schlüssel »out« die gerade aufgeschnappte Stdout-Zeile des Kindprozesses parat hält (Zeilen 332 und 334). Leider feuert »PoCo::Player::Musicus« keinen Event ab, wenn ein gespieltes Stück beendet ist.

Also muss Rateplay den Player in kurzen Abständen anstoßen und die aktuelle Liedposition mit »getpos()« abfragen. Kommt ein negativer Wert zurück, dreht Musicus gerade Däumchen und lechzt nach neuen Songs. Die Funktion implementiert ab Zeile 69 dieses periodische Pollen. Sie ist dem »poll_player«-Event zugewiesen und schickt eine »getpos()«-Abfrage an Musicus. Als Ergebnis dieser Abfrage geht ein »getpos«-Event an die »main«-Session. Danach sorgt Zeile 71 dafür, dass der Kernel nach einer Sekunde wieder einen »poll_player«-Event auslöst. Die ab Zeile 79 definierte Funktion »getpos()« frischt die Our-Variable

Listing 1: »rateplay«

```
001 #!/usr/bin/perl
002 ###########################################
003 # rateplay - Rate MP3s and play them
004 # Mike Schilli, 2004 (m@perlmeister.com)
005 ###########################################
006 use strict; use warnings;
007
008 our $DB_NAME  = "/data/rp.dat";
009 our $SONG_DIR = "/ms1/SONGS/pods";
010 our $FIND     = "/usr/bin/find";
011
012 use Gtk; use POE;
013 use Class::DBI;
014 use POE::Component::Player::Musicus;
015 use Algorithm::Numerical::Shuffle
016                                qw(shuffle);
017 my (%GUI, %RATED, $TAG, $SONG, @PLAYLIST,
018     @MP3S);
019 my @PLAY_ENERG = (0, 0, 0, 0, 0);
020 my @PLAY_SCHMO = (0, 0, 0, 0, 0);
021 my $RATE_ENERG = 0;
022 my $RATE_SCHMO = 0;
023 my @RATE_ENERG_BUTTONS = ();
024 my @RATE_SCHMO_BUTTONS = ();
025
026 ###########################################
027 package Rateplay::DBI;
028 ###########################################
029 use base q(Class::DBI);
030 use Class::DBI::AbstractSearch;
031
032 __PACKAGE__->set_db('Main',
033   "dbi:SQLite:$main::DB_NAME", 'root', '');
034
035 if(! -e "$main::DB_NAME") {
036     __PACKAGE__->set_sql(create => q{
037         CREATE TABLE rated_songs (
038             path VARCHAR(256)
039                  PRIMARY KEY NOT NULL,
040             energize INT, schmoop  INT
041         )
042     });
043     __PACKAGE__->sql_create()->execute();
044 }
045
046 ###########################################
047 package Rateplay::Song;
048 ###########################################
049 use base q(Rateplay::DBI);
050
051 __PACKAGE__->table('rated_songs');
052 __PACKAGE__->columns(
053     All => qw(path energize schmoop));
054
055 ###########################################
056 package main;
057 ###########################################
058
059 my $PLAYER =
060     POE::Component::Player::Musicus->new();
061
062 POE::Session->create(
063    package_states => [ "main" => [
064      qw(getpos getinfocurr mp3_stdout
065         song   scan_mp3s)]],
066
067    inline_states => {
068      _start      => \&my_gtk_init,
069      poll_player => sub {
070       $PLAYER->getpos();
071       $poe_kernel->delay('poll_player', 1);
072    }});
073
074 $poe_kernel->post("main", "poll_player");
075 $poe_kernel->post("main", "scan_mp3s");
076 $poe_kernel->run();
077
078 ###########################################
079 sub getpos {
080 ###########################################
081   our $POS;
082
083   next_in_playlist() if defined $POS and
084        $POS > 0 and $_[ARG0] < 0;
085   $POS = $_[ARG0];
086 }
087
088 ###########################################
089 sub getinfocurr {
090 ###########################################
091     $TAG = $_[ARG0];
092     $GUI{artist}->set($TAG->{artist});
093     $GUI{title}->set($TAG->{title});
094 }
095
096 ###########################################
097 sub song {
098 ###########################################
099     $SONG = $_[ARG0];
100     $PLAYER->stop();
101     $PLAYER->play($SONG);
102     $PLAYER->getinfocurr();
103     update_rating($SONG);
104 }
105
106 ###########################################
107 sub scan_mp3s {
108 ###########################################
109   %RATED = map { $_->path() => 1 }
110            Rateplay::Song->retrieve_all();
111
112   my $comp = POE::Component::Child->new(
113     events => { 'stdout' => 'mp3_stdout' },
114   );
115
116   $comp->run($FIND, $SONG_DIR);
117 }
[...]
120 sub add_label {
[...]
137 ###########################################
138 sub my_gtk_init {
139 ###########################################
140   my @btns = ("Play Rated", "Play Next",
141           "Play Previous", "Random Rate");
142
143   $poe_kernel->alias_set('main');
144
145   $GUI{mw} = Gtk::Window->new();
146   $GUI{mw}->set_default_size(150,200);
147
148   $GUI{vb} = Gtk::VBox->new(0, 0);
149
150   $GUI{$_}= Gtk::Button->new($_) for @btns;
151
152   my $tbl = Gtk::Table->new(2, 6);
153   $GUI{vb}->pack_start($tbl, 1, 1, 0);
154
155   add_label($tbl, 'Energize', 0, 1, 0, 1);
156   add_buttons($tbl,
157     sub { $PLAY_ENERG[$_[1]] ^= 1 }, 0);
158   add_label($tbl, 'Schmoop', 0, 1, 1, 2);
159   add_buttons($tbl,
160     sub { $PLAY_SCHMO[$_[1]] ^= 1}, 1);
161
162         # Status line on top of buttons
163   $GUI{status} = add_label($GUI{vb}, "");
164
165       # Pack buttons
166   $GUI{vb}->pack_start($GUI{$_}, 0, 0, 0)
167                                   for @btns;
168
169   for(qw(artist title)) {
```

»$POS« auf, die einen Integer-Wert speichert; er entspricht der aktuellen Position im Song.

War der vorherige Wert größer null und kommt nun ein negativer Wert daher, ist dies ein Indiz dafür, dass der Player gerade einen Song beendet hat und Zeile 83 die »next_in_playlist()«-Funktion aufrufen soll. Diese extrahiert das erste Element des globalen Array »@PLAYLIST«, hängt es hinten wieder an und reicht es in Zeile 307 dem Player zum Abspielen weiter. Falls der erste Parameter von »next_in_playlist()« gesetzt ist, geht die nächste Fahrt rückwärts und der vorige Song wird gespielt.

Ist das Resultat aufgrund kompensierenden Vor- und Rückwärtsfahrens wieder derselbe Song, fährt Zeile 302 nochmals eins weiter. Für jeden neu gespielten Song ruft »song()« die ab Zeile 311 definierte Funktion »update_rating()« auf. Sie sieht mittels der »search()«-Methode in der Datenbank nach, ob der Song schon bewertet ist, und besetzt die Radiobuttons mit den entsprechenden Werten für Energize und Schmoop, falls sie ein Ergebnis findet. Falls nicht, stellt sie die kleinstmöglichen Werte ein. So sieht der Benutzer für jeden Song auf einer

Listing 1: »rateplay« (Fortsetzung)

```
170     $GUI{$_} = add_label($GUI{vb}, "");
171   }
172
173   $GUI{rate_table} = Gtk::Table->new(2, 6);
174   $GUI{vb}->pack_start($GUI{rate_table},
175                                  0, 0, 0);
176
177   add_label($GUI{rate_table},
178             'Energize', 0, 1, 0, 1);
179   attach_radio_buttons($GUI{rate_table},
180     sub { $RATE_ENERG = $_[1]+1;
181         }, 0, \@RATE_ENERG_BUTTONS);
182   add_label($GUI{rate_table},
183             'Schmoop', 0, 1, 1, 2);
184   attach_radio_buttons($GUI{rate_table},
185     sub { $RATE_SCHMO = $_[1]+1;
186         }, 1, \@RATE_SCHMO_BUTTONS);
187
188   my $rate  = Gtk::Button->new('Rate');
189   $GUI{vb}->pack_start($rate,  0, 0, 0);
190   $GUI{mw}->add($GUI{vb});
191
192       # Destroying window
193   $GUI{mw}->signal_connect('destroy',
194     sub {Gtk->exit(0)});
195
196         # Pressing Play Rated button
197   $GUI{$btns[0]}->signal_connect('clicked',
198     sub { @PLAYLIST = select_songs();
199       $GUI{status}->set("Playlist has " .
200         scalar @PLAYLIST . " songs.");
201       next_in_playlist();
202     });
203
204       # Pressing Play Next button
205   $GUI{$btns[1]}->signal_connect('clicked',
206     sub { next_in_playlist() });
207
208       # Pressing Play Previous button
209   $GUI{$btns[2]}->signal_connect('clicked',
210     sub { next_in_playlist(1) });
211
212       # Pressing Random Rate Button
213   $GUI{$btns[3]}->signal_connect('clicked',
214     sub { @PLAYLIST = shuffle @MP3S;
215       $GUI{status}->set("Random Rating " .
216               scalar @PLAYLIST . " songs.");
217       next_in_playlist();
218     });
219         # Pressing Rate button
220   $rate->signal_connect('clicked',
221       sub { return unless defined $TAG;
222             process_rating();
223             next_in_playlist();
224           } );
225
226     $GUI{mw}->show_all();
227 }
228
229 ###########################################
230 sub add_buttons {
231 ###########################################
232     my($table, $sub, $row) = @_;
233
234     for (0..4) {
235       my $b = Gtk::CheckButton->new($_+1);
236       $b->signal_connect(clicked=> $sub, $_);
237       $table->attach_defaults($b, 1+$_, 2+$_,
238                               0+$row, 1+$row);
239     }
240 }
[...]
243 sub attach_radio_buttons {
[...]
260 ###########################################
261 sub process_rating {
262 ###########################################
263     my $rec = Rateplay::Song->find_or_create(
264                     { path     => $SONG });
265
266     $rec->energize($RATE_ENERG);
267     $rec->schmoop($RATE_SCHMO);
268     $rec->update();
269 }
270
271 ###########################################
272 sub select_songs {
273 ###########################################
274     my @energ = grep { $PLAY_ENERG[$_-1] }
275            (1..@PLAY_ENERG);
276     my @schmo = grep { $PLAY_SCHMO[$_-1] }
277            (1..@PLAY_SCHMO);
278
279     return sort { rand > 0.5 }
280            map { $_->path() }
281                Rateplay::Song->search_where({
282                  energize => [@energ, 0],
283                  schmoop  => [@schmo, 0]},
284     );
285 }
286
287 ###########################################
288 sub next_in_playlist {
289 ###########################################
290     my($backward) = @_;
291
292     return unless scalar @PLAYLIST;
293     my $path;
294
295     { if($backward) {
296         $path = pop @PLAYLIST;
297         unshift @PLAYLIST, $path;
298       } else {
299         $path = shift @PLAYLIST;
300         push @PLAYLIST, $path;
301       }
302       redo if defined $SONG and
303             $SONG eq $path and @PLAYLIST > 1;
304     }
305
306     $PLAYER->stop();
307     $poe_kernel->post('main', 'song', $path);
308 }
309
310 ###########################################
311 sub update_rating {
312 ###########################################
313     my ($path) = @_;
314
315     if(my ($song) = Rateplay::Song->search(
316                         path => $path)) {
317       my $e = $song->energize();
318       my $s = $song->schmoop();
319       $RATE_SCHMO_BUTTONS[$s-1]->activate();
320       $RATE_ENERG_BUTTONS[$e-1]->activate();
321     } else {
322       $RATE_SCHMO_BUTTONS[0]->activate();
323       $RATE_ENERG_BUTTONS[0]->activate();
324     }
325 }
326
327 ###########################################
328 sub mp3_stdout {
329 ###########################################
330     my ( $self, $args ) = @_[ ARG0 .. $#_ ];
331
332     return if exists $RATED{$args->{out}};
333
334     push @MP3S, $args->{out};
335
336     $GUI{status}->set(scalar @MP3S .
337                " songs ready for rating.");
338 }
```

Playlist das Rating und korrigiert es gegebenenfalls. Dazu genügt es, die gewünschten Werte einzustellen und auf »Rate« zu klicken.

Geschmackvolle Auswahl

Die ab Zeile 272 definierte Funktion »select_songs()« kümmert sich um die Auswahl von Songs für eine Playlist gemäß den im GUI eingestellten Checkbutton-Werten für Energize und Schmoop. Die Arrays »@PLAY_ENERG« und »@PLAY_SCHMO« führen jeweils fünf Elemente. Ist der zugeordnete Checkbutton am oberen Ende des GUI aktiviert, enthält das entsprechende Element den Wert 1, andernfalls 0. Steht in »@PLAY_ENERG« zum Beispiel »(0,0,1,1,0)«, sind der dritte und der vierte Checkbutton in der Energize-Leiste aktiviert, alle anderen dagegen nicht.

Zeile 274 extrahiert daraus die Liste der gewünschten Energize-Werte und legt sie in »@energ« ab. Der »search_where()«-Aufruf in Zeile 281 fügt noch einen ungültigen Null-Wert hinzu, um zu verhindern, dass es zu einem Fehler kommt, falls »@energ« leer ist. Denn »search_where()« reagiert auf leere Arrays allergisch. Beide Kriterien für Energize und Schmoop verknüpft »search_where()« mit einem logischen UND, also äquivalent zu »WHERE a AND b« in SQL. Die Elementwerte der übergebenen Arrays hingegen werden als ODER ausgewertet. So sortiert folgender Code alle Songs in den gegebenen Geschmacksgrenzen:

```
Rateplay::Song->search_where({
    energize => [2, 3, 0],
    schmoop  => [1, 0]});
```

Die entsprechende SQL-Abfrage sieht folgendermaßen aus:

```
SELECT * from rated_songs
WHERE energize = 2 OR
      energize = 3 OR
      energize = 0
AND   schmoop = 1 OR
      schmoop = 0
```

Das vor den »map()«-Befehl gesetzte »sort { rand < 0.5 }« in Zeile 279 wirbelt die Ergebnisliste durcheinander, bevor sie an den Player geht – schließlich will der Benutzer Abwechslung und nicht immer dieselbe Reihenfolge.

Die Funktion »process_rating()« ab Zeile 261 sucht mit »find_or_create()« einen Eintrag unter dem angegebenen MP3-Pfad in der Datenbank. Sie gibt als Ergebnis das gefundene Objekt zurück. Falls sie keins findet, erzeugt »find_or_create()« einfach einen neuen Eintrag. Die »energize()«- und »schmoop()«-Methodenaufrufe setzen die entsprechenden Felder des Datensatzes und die »update()«-Methode schreibt alles in die Datenbank zurück.

Äußerlichkeiten

Die ab Zeile 138 definierte Funktion »my_gtk_init()« baut die GTK-Oberfläche auf. Alle GUI-Objekte landen unter beschreibenden Namen im globalen Hash »%GUI«, damit sie schön gruppiert und einfach global abrufbar sind. Denn manche Funktion muss in bestimmten Situationen schnell einige der grafischen Elemente auffrischen. Wie in **[2]** kommen auch hier zwei verschiedene GUI-Container zum Einsatz, nämlich »Gtk::VBox« und »Gtk::Table«, beide mit verschiedenen Pack-Verfahren (»pack_start()« und »attach_defaults()«).

Die Funktion »add_buttons()« ab Zeile 230 wird für beide Checkbox-Buttonreihen an der oberen Hälfte des GUI aufgerufen. Jedes Mal schiebt das Hauptprogramm eine andere Funktion hinein, die aufgerufen wird, falls der Benutzer den Button anklickt. Ab Zeile 193 definiert Rateplay, welche Aktionen auf Maus-Events folgen. Als Reaktion auf ein »destroy«-Signal (das eintrifft, falls der Benutzer das Applikationsfenster schließt), reißt »Gtk->exit(0)« das GUI ab.

Der Button »Play Rated« (»$btns[0]«) löst »select_songs()« aus und stößt mit »next_in_playlist()« die Playliste an. »Play Next« und »Play Previous« fahren vor und zurück und »Random Rate« (»$btns[3]«) wirbelt die im globalen Array »@MP3S« gespeicherten unbewerteten MP3s mittels der »shuffle«-Funktion aus »Algorithm::Numerical::Shuffle« durcheinander, um sie dann der Reihe nach anzubieten.

Die Callback-Funktion des »Rate«-Buttons schließlich greift auf die in der Funktion »getinfocurr()« gesetzte globale Variable »$TAG« zu, die das MP3-Tag des gerade abgespielten Songs enthält, und ruft »process_rating()« auf, um einen Datenbankeintrag für den Song entsprechend den eingestellten Radiobuttons vorzunehmen.

Installation

Damit Rateplay mit dem MP3-Player zusammenspielt, muss Xmms auf dem Rechner installiert sein. Danach lädt der Anwender die Musicus-Quellen von **[3]** herunter, entpackt sie und tippt »make« ein. Das Binary »musicus« kopiert er dann nach »/usr/bin/«.

Die Perl-Module »POE«, »PoCo::Player::Musicus« und »Gtk« finden sich im CPAN. Der Artikel unter **[2]** gibt einige Hinweise zur Installation. Rateplay benötigt auch die Module »DBI«, »DBD::SQLite«, »Class::DBI« »Class::DBI::AbstractSearch« und »Algorithm::Numerical::Shuffle«. Die CPAN-Shell löst alle eventuell auftretenden Abhängigkeiten automatisch auf.

Die Programmierer von Musicus und »POE::Component::Player::Musicus« arbeiten eifrig an Verbesserungen ihrer Programme. Falls die aktuellen Versionen nicht zusammenarbeiten wollen, stehen unter **[6]** zwei Tarbälle bereit, die funktionieren. *(mwe)* ■

Infos

[1] Listings zu diesem Artikel: [ftp://ftp.linux-magazin.de/pub/listings/magazin/2004/07/Perl]
[2] Michael Schilli, „Kurs-Makler“: Linux-Magazin 04/04, S. 116
[3] Musicus-Homepage: [http://muth.org/Robert/Musicus]
[4] Michael Schilli, „Musik aus dem Keller“: Linux-Magazin, 03/03, S. 102
[5] SQLite: [http://sqlite.org]
[6] Musicus und sein POE-Pendant: [http://perlmeister.com/musicus]

Der Autor

Michael Schilli arbeitet als Software-Engineer für AOL/Netscape in Mountain View, Kalifornien. Er hat „Goto Perl 5“ (deutsch) und „Perl Power“ (englisch) für Addison-Wesley geschrieben und ist unter [mschilli@perlmeister.com] zu erreichen und seine Homepage heißt [http://perlmeister.com].