Categorizing MP3s and creating playlists

DJ Training

Depending on their current mood, music fans may fancy a bit of rock, or some easy listening pop. An MP3 player with a GTK-based graphical interface selects tracks to match your mood, creates a playlist, and plays the tracks. The Perl Object Environment keeps everything running smoothly.

BY MICHAEL SCHILLI

Every large collection of MP3s ripped from CDs has undreamt of treasures slumbering in its depths. A computer-aided selector can dig those hidden treasures out from among thousands of MP3 files, allowing for amazing compilations based on simple criteria.

The Perl script we will be looking at in this month's column, *rateplay*, plays the user a selection of tracks in random order. The user then rates the tracks according to two criteria: the energize factor, and the schmoop factor. The energize factor describes how lively a track is, and schmoop how laid back. On a scale from one to five, "Thunderstruck" by AC/DC might have an energize factor of five and a schmoop factor of one. "Don't Know Why" by Norah Jones would have an energize factor of one and a schmoop factor of five.

Figure 1: Rateplay playing songs with energize levels between three and five and schmoop levels one to three. The current track is a heavy metal song with an energize level of five and a schmoop value of one.

Whenever a user rates a track, the path to the song and both factors are stored in a database. After collecting a number of evaluations the script can create and play a playlist in response to a request such as "play a few fast tracks, but don't scare my girlfriend off".

GUI-based Interface

Figure 1 shows the script in action. Music fans just need to select acceptable energize and schmoop factors in the two upper button bars to play tracks they have already rated. Clicking on *Play Rated* creates a playlist using songs with matching ratings from the database, and goes on to play the list in random order. You can select *Play Next* and *Play Previous* to jump to the next track or back to the previous track.

To rate new tracks, users simply click the *Random Rate* button. Rateplay will then create a playlist of previously unrated songs and play the list. While this is happening, users can set the levels for each track using the button bar at the bottom of the window. You can assign only one energize and one schmoop level per track. Clicking on *Rate* stores the values in the database and tells Rateplay to move on to the next file.

By Twos and Threes

Rateplay uses several Perl modules. In addition to the popular POE and GTK modules [2] for a smooth GUI-based interface, we will be using the Musicus [3] command line player by Robert Muth, a C++ program based on the dynamic libraries of the Xmms player. The *POE::Component::Player::Musicus* module (this is abbreviated to *PoCo::Player::Musicus* in the following sections) by Curtis Hawthorne integrates the MP3 player into the POE environment, allowing the GUI to remote-control the player smoothly.

Rateplay uses the object-oriented *Class::DBI* abstraction to store ratings, using an SQLite database under the hood (see [4]). SQLite can generate a professional database in a single file. Of course the DBI series at CPAN has a Perl module to match: *DBD::SQLite*. In fact, SQLite is a plain old SQL database. To discover how many rated songs Rateplay has in its *rated_songs* table, users can run the command line *sqlite* tool to access the *rp.dat* database file created by Rateplay, and issue the following SQL command:

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

In our example, there are 887 rated songs. Enough tracks to generate amazingly cool playlists!

Rateplay in Detail

The Rateplay program is quite extensive. Listing 1 shows the source code, which you can also download from [1]. The configuration lines, 10 through 13, define the path to the database file (using the *$DB_NAME* variable to do so), and the

directory (in *$SONG_DIR*), in which the *find* program will search recursively for files ending in *.mp3*.

The global arrays, *@PLAY_ENERG* and *@PLAY_SCHMO*, contain the values for the song selection checkboxes at the top of the GUI. In contrast to this, the scalar values of *$RATE_ENERG* and *$RATE_SCHMO* reflect the state of the radio buttons at the bottom of the window, and expect values between one and five for the energize and schmoop factors. The *@RATE_ENERG_BUTTONS* and *@RATE_SCHMO_BUTTONS* arrays contain the radio button objects as array elements, allowing the GUI to set the values stored in the database for a track.

The *Rateplay::DBI* class in line 33 ff. inherits from *Class::DBI* and defines the object-oriented abstraction of the SQLite database. If the database does not exist (in SQLite, this is indicated by the fact that the corresponding file does not exist), the SQL code in line 47 ff. creates the database file, and the *rated_songs* table with the columns *path* (path to MP3 file), *energize* (for the energize level) and *schmoop* (schmoop level). The *execute* method in line 55 actually makes it happen.

Pulling in *Class::DBI::AbstractSearch* in line 37 adds extended queries to what *Class::DBI* already provides for a class derived from it. Later on, this will be put to action via *Rateplay::Song->search_where()*, which executes a SQL statement with a WHERE clause. The *Rateplay::Song* class in line 59 ff. defines the OO abstraction of the table *rated_songs*. Isn't it nice to have the rest of the script 100% free of any SQL statements?

Using POE to Control the MP3 Player

The main program is contained in the *main* package starting in line 70. It defines the POE session, which runs the GUI and the player. The array referenced by the *package_states* parameter creates a number of functions which are defined later in the script and called by POE events with the same names. For example, whenever the main program calls the player's *getpos()* method, the player responds with the position in the current track by sending a *getpos* event to the main POE session. The *package_states* reference just mentioned tells the main session to jump to the *getpos()* function defined in line 96 ff. in this case. Figure 2 shows you what the complete session looks like and which discrete states it consists of.

A similar thing happens with *getinfocurr*. According to the *PoCo::Player::Musicus* documentation, if someone calls the player object's *getinfocurr()* method, it will call back into the main session, passing artist, track name, and some MP3 tag information on the current track. Lines 111 and 113 in the callback function *getinfocurr* update the artist and track name display in the GUI.

Whenever the player needs to play a new track, Rateplay sends a *song* event to the *main* session, like the one shown in line 415. The *song* event in turn has been defined to call the *song()* function shown in lines 118 ff.. It grabs the path to the MP3 file as POE's first argument ARG0, then stops the player and immediately points it at the new MP3 file to be played.

The *scan_mp3s* event is triggered in line 91 shortly after the system launch; it tells the script to jump to the *scan_mp3s* function defined in line 128. *scan_mp3s* calls *retrieve_all()* to retrieve all rated songs from the database and stores them as keys in the global hash *%RATED*. It then goes on to spawn a child process in a *PoCo::Child* session; the child process calls the external *find* command to discover MP3 files on the hard disk. When find discovers a file, it writes the path to stdout.

The session then follows the event definition in line 139 (and the *package_states* definition in line 79) and jumps to the *mp3_stdout()* function, which is defined in line 444 and following. It appends the filename to the global array *@MP3S*, if the user has not yet rated the file. Line 455 updates the status display for the current search. As described in [2], POE uses unusual parameter constants. For example, *ARG0* is a constant holding the index for the position of the first parameter in *@_* passed to the

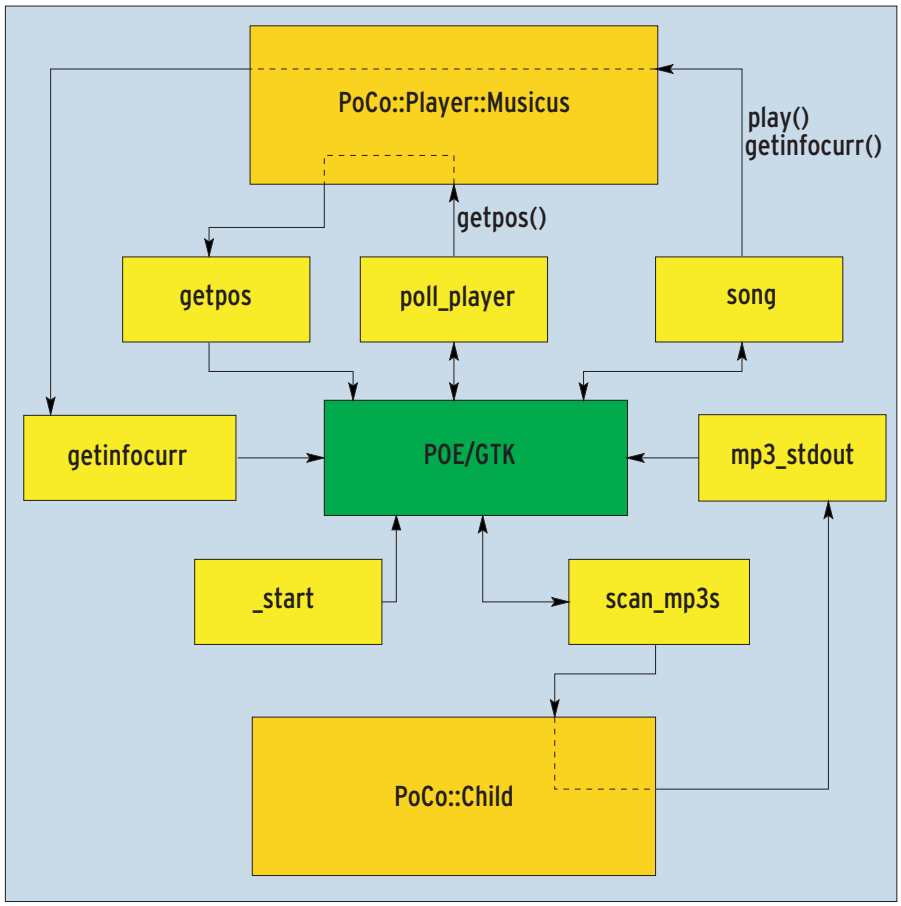

Figure 2: POE makes the program jump between various states. The *PoCo* processes run in parallel. Double arrows indicate temporary transitions to states interacting with separated processes.

event. When *PoCo::Child* enters the callback, the first parameter is a reference to a hash that contains the stdout line grabbed by the child process. Lines 451 and 453 show how to dig it up under the hash's key *out*.

Unfortunately, *PoCo::Player::Musicus* does not trigger an event when the player finishes playing a track. This means that Rateplay has to query the player at regular intervals, using *getpos()* to discover the current position within the track. If a negative value is returned, this indicates that Musicus is idling, ready to play new songs. To catch this, the anonymous function defined in line 83 and following implements periodic polling. It is bound to the *poll_player* event and sends a *getpos()* request to Musicus. In response to the request, the Musicus POE component sends a *getpos* event back to the *main* session. To close the polling loop, line 85 tells the kernel to trigger the *poll_player* event again one second later. The *getpos()* callback function defined in line 96 ff. updates the global *$POS* variable which stores an integer value for the current position within the current song.

If the previous value of *$POS* was positive, and the current value is negative, it is safe to assume that the player has just finished playing a track, and thus needs to call the *next_in_playlist()* function defined in line 390. This function extracts the first element in the global array *@PLAYLIST*, moves the element to the end of the list, and passes it to the player for output in line 415. In contrast, if *next_in_playlist()* is passed a parameter holding a true value, the script will go backwards and play the previous song instead.

If the result is the same song, due to quickly skipping back and forward, line 408 moves one step further. For each new song that is played, *song()* calls the function defined in line 420 and following, *update_rating()*. It uses the *search()* method to check the database for a song rating, and set the radio buttons accord-

Listing 1: *rateplay*

```
001 #!/usr/bin/perl
002 ##################################
003 # rateplay - Rate & Play MP3s
004 # Mike Schilli, 2004
005 # (m@perlmeister.com)
006 ##################################
007 use strict;
008 use warnings;
009
010 our $DB_NAME ="/data/rp.dat";
011 our $SONG_DIR=
012             "/ms1/SONGS/pods";
013 our $FIND = "/usr/bin/find";
014
015 use Gtk;
016 use POE;
017 use Class::DBI;
018 use
POE::Component::Player::Musicus;
019 use
Algorithm::Numerical::Shuffle
qw(shuffle);
020
021 my (%GUI, %RATED, $TAG,
022     $SONG, @PLAYLIST, @MP3S);
023 my @PLAY_ENERG =
024   ( 0, 0, 0, 0, 0 );
025 my @PLAY_SCHMO =
026   ( 0, 0, 0, 0, 0 );
027 my $RATE_ENERG          = 0;
028 my $RATE_SCHMO          = 0;
029 my @RATE_ENERG_BUTTONS = ();
030 my @RATE_SCHMO_BUTTONS = ();
031
032 ##################################
033 package Rateplay::DBI;
034 ##################################
035 use base q(Class::DBI);
036 use
037   Class::DBI::AbstractSearch;
038
039 __PACKAGE__->set_db(
040   'Main',
041 "dbi:SQLite:$main::DB_NAME",
042   'root', '');
043
044 if ( !-e "$main::DB_NAME" ) {
045   __PACKAGE__->set_sql(
046     create => q{
047   CREATE TABLE rated_songs (
048     path VARCHAR(256)
049        PRIMARY KEY NOT NULL,
050     energize INT,
051     schmoop  INT
052   )});
053
054   __PACKAGE__->sql_create()
055     ->execute();
056 }
057
058 ##################################
059 package Rateplay::Song;
060 ##################################
061 use base q(Rateplay::DBI);
062
063 __PACKAGE__->table(
064   'rated_songs');
065 __PACKAGE__->columns( All =>
066     qw(path energize schmoop)
067 );
068
069 ##################################
070 package main;
071
072 my $PLAYER =
POE::Component::Player::Musicus-
>new();
073
074 POE::Session->create(
075   package_states => [
076     "main" => [
077       qw(getpos getinfocurr
078          mp3_stdout song
079          scan_mp3s) ]],
080
081   inline_states => {
082     _start => \&my_gtk_init,
083     poll_player => sub {
084       $PLAYER->getpos();
085       $poe_kernel->delay(
086         'poll_player', 1 );
087       }});
088
089 $poe_kernel->post( "main",
090   "poll_player" );
091 $poe_kernel->post( "main",
092   "scan_mp3s" );
093 $poe_kernel->run();
094
095 ##################################
096 sub getpos {
097 ##################################
098   our $POS;
099
100   next_in_playlist()
101     if defined $POS
102     and $POS > 0
103     and $_[ARG0] < 0;
104   $POS = $_[ARG0];
105 }
106
107 ##################################
108 sub getinfocurr {
109 ##################################
110   $TAG = $_[ARG0];
111   $GUI{artist}
112     ->set( $TAG->{artist} );
```

ing to the energize and schmoop values found. If no ratings are present, it displays the smallest possible values. Thus, while playing a rated list, the user sees a rating for each song, and can correct it if needed. To do so, users simply set the desired values and click on *Rate*.

Good Taste

The function *select_songs()* defined in line 369 ff. selects tracks and composes a playlist based on the checkbox values for energize and schmoop set in the GUI. The *@PLAY_ENERG* and *@PLAY_SCHMO* arrays each contain five elements. If the corresponding checkbox at the top of the GUI is checked, the element has a value of 1; if not, it has a value of 0. Let's assume that *@PLAY_ENERG* contains *(0,0,1,1,0)*; this indicates that checkboxes number three and four are checked, and the others are not.

Line 371 extracts the desired energize values from the array and stores them in *@energ*. The call to *search_where()* in line 382 adds an additional (and invalid) value of zero, to prevent *search_where()* from acting up if the *@energ* array is empty. *search_where()* uses a logical AND to link both criteria for energize and schmoop; this is equivalent to *WHERE a AND b* in SQL. In contrast to this, the element values in the arrays passed to the function are ORed. Thus, the following code will sort songs to reflect the user's taste:

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

The corresponding SQL request looks like this:

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

Listing 1: *rateplay*

```
113   $GUI{title}
114     ->set( $TAG->{title} );
115 }
116
117 ################################
118 sub song {
119 ################################
120   $SONG = $_[ARG0];
121   $PLAYER->stop();
122   $PLAYER->play($SONG);
123   $PLAYER->getinfocurr();
124   update_rating($SONG);
125 }
126
127 ################################
128 sub scan_mp3s {
129 ################################
130   %RATED =
131     map { $_->path() => 1 }
132       Rateplay::Song
133       ->retrieve_all();
134
135   my $comp =
136     POE::Component::Child
137     ->new(
138     events => {
139      'stdout' => 'mp3_stdout'
140     });
141
142   $comp->run($FIND,
143              $SONG_DIR);
144 }
145
146 ################################
147 sub add_label {
148 ################################
149   my ($parent, $text,
150       @coords) = @_;
151
152   my $lbl= Gtk::Label->new();
153   $lbl->set_alignment(
154                     0.5, 0.5);
155   $lbl->set($text);
156
157   if (ref $parent eq
158       "Gtk::Table") {
159     $parent->attach_defaults(
160       $lbl, @coords);
161   } else {
162     $parent->pack_start(
163       $lbl, 0, 0, 0);
164   }
165
166   return $lbl;
167 }
168
169 ################################
170 sub my_gtk_init {
171 ################################
172   my @btns = (
173    "Play Rated", "Play Next",
174    "Play Previous",
175    "Random Rate"
176   );
177
178   $poe_kernel->alias_set(
179     'main');
180
181   $GUI{mw} =
182     Gtk::Window->new();
183   $GUI{mw}->set_default_size(
184                     150, 200);
185
186   $GUI{vb} =
187     Gtk::VBox->new(0, 0);
188
189   $GUI{$_} =
190     Gtk::Button->new($_)
191       for @btns;
192
193   my $tbl =
194     Gtk::Table->new(2, 6);
195   $GUI{vb}->pack_start(
196               $tbl, 1, 1, 0);
197
198   add_label($tbl,
199     'Energize', 0, 1, 0, 1);
200   add_buttons(
201     $tbl, sub {
202      $PLAY_ENERG[$_[1]] ^= 1;
203     }, 0);
204   add_label( $tbl, 'Schmoop',
205     0, 1, 1, 2 );
206   add_buttons(
207     $tbl, sub {
208      $PLAY_SCHMO[$_[1]] ^= 1;
209     }, 1);
210
211   # Status on top of buttons
212   $GUI{status} =
213     add_label($GUI{vb}, "");
214
215   # Pack buttons
216   $GUI{vb}->pack_start(
217     $GUI{$_}, 0, 0, 0)
218       for @btns;
219
220   for (qw(artist title)) {
221     $GUI{$_} = add_label(
222                 $GUI{vb}, "");
223   }
224
225   $GUI{rate_table} =
226     Gtk::Table->new(2, 6);
227   $GUI{vb}->pack_start(
228     $GUI{rate_table},0,0,0);
229
```

The *sort { rand < 0.5 }* statement in line 379 before the *map()* command, mixes up the results before sending them to the player – after all, users want a little variety rather than the same playing order every time.

The *process_rating()* function in line 355 and following uses *Class::DBI*'s *find_or_create()* method to search for an entry matching the specified MP3 path in the database. It returns the object it finds. If it fails to find any matching object, *find_ or_create()* simply creates a new entry. The *energize()* and *schmoop()* methods set the corresponding database fields, and *update()* then writes the results back to the database.

Appearances

The *my_gtk_init()* function defined in line 170 and following sets up the GTK interface. All the GUI objects are stored under mnemonic names in a global hash called *%GUI*. This groups them nicely and ensures that they can be accessed globally. Some functions will need to refresh their graphical elements in a hurry in certain situations. As in [2], we again use two different types of GUI containers: that is *Gtk::VBox* and *Gtk::Table*, which require us to use different packing procedures (*pack_start()* and *attach_defaults()*).

add_buttons() in line 312 and following is called for both rows of checkboxes in the top half of the GUI. The main program passes in a reference to a different callback function each time, which will be called when the user clicks the corresponding button. Rateplay defines the actions for mouse events in line 254 and following lines. The reaction to the *destroy* signal (which occurs if the user closes the application window), for example, is to call *Gtk->exit(0)* and kill the GUI.

The *Play Rated* (*$btns[0]*) button triggers *select_songs()*, and uses *next_in_playlist()* to play the next song. *Play Next* and *Play Previous* skip forward and back, and *Random Rate* (*$btns[3]*) calls the *shuffle* function from *Algorithm::Numeri-*

Listing 1: *rateplay*

```
230   add_label(
231     $GUI{rate_table},
232     'Energize', 0, 1, 0, 1);
233   attach_radio_buttons(
234     $GUI{rate_table}, sub {
235       $RATE_ENERG = $_[1] +1;
236     }, 0,
237     \@RATE_ENERG_BUTTONS);
238   add_label(
239     $GUI{rate_table},
240     'Schmoop', 0, 1, 1, 2);
241   attach_radio_buttons(
242     $GUI{rate_table}, sub {
243       $RATE_SCHMO = $_[1] +1;
244     }, 1,
245     \@RATE_SCHMO_BUTTONS);
246
247   my $rate =
248     Gtk::Button->new('Rate');
249   $GUI{vb}->pack_start(
250     $rate, 0, 0, 0);
251   $GUI{mw}->add($GUI{vb});
252
253   # Destroying window
254   $GUI{mw}->signal_connect(
255     'destroy',
256     sub { Gtk->exit(0) } );
257
258 # Pressing Play Rated button
259   $GUI{ $btns[0] }
260     ->signal_connect(
261       'clicked', sub {
262         @PLAYLIST =
263           select_songs();
264         $GUI{status}->set(
265             "Playlist has "
266           . scalar @PLAYLIST
267           . " songs." );
268         next_in_playlist();
269       });
270
271   # Pressing Play Next button
272   $GUI{ $btns[1] }
273     ->signal_connect(
274       'clicked', sub {
275         next_in_playlist();
276       });
277
278   # Pressing "Play Previous"
279   $GUI{ $btns[2] }
280     ->signal_connect(
281       'clicked', sub {
282         next_in_playlist(1);
283       });
284
285   # Pressing "Random Rate"
286   $GUI{ $btns[3] }
287     ->signal_connect(
288       'clicked', sub {
289         @PLAYLIST =
290           shuffle @MP3S;
291         $GUI{status}->set(
292           "Random Rating "
293           . scalar @PLAYLIST
294           . " songs." );
295         next_in_playlist();
296       });
297
298   # Pressing "Rate" button
299   $rate->signal_connect(
300     'clicked', sub {
301       return
302         unless defined $TAG;
303       process_rating();
304       next_in_playlist();
305     }
306   );
307
308   $GUI{mw}->show_all();
309 }
310
311 ################################
312 sub add_buttons {
313 ################################
314   my($table, $sub, $row)= @_;
315
316   for (0 .. 4) {
317     my $b =
318       Gtk::CheckButton->new(
319         $_ + 1);
320     $b->signal_connect(
321         clicked => $sub, $_);
322     $table->attach_defaults(
323       $b, 1 + $_, 2 + $_,
324       0 + $row, 1 + $row );
325   }
326 }
327
328 ################################
329 sub attach_radio_buttons {
330 ################################
331   my ($table, $sub, $row,
332       $buttons) = @_;
333
334   my $group;
335
336   for (0 .. 4) {
337     my $btn =
338       Gtk::RadioButton->new(
339       $_ + 1,
340       defined $group
341       ? $group : ());
342     $group = $btn;
343     $btn->signal_connect(
344       clicked => $sub, $_);
345     push @$buttons, $btn;
346     $table->attach_defaults(
```

cal::Shuffle to randomize the order of the non-rated MP3s stored in the global *@MP3S* array, so that the program can offer them to the user for rating one by one.

Finally, the callback function for the *Rate* button accesses the *$TAG* variable set in the *getinfocurr()*, which contains the MP3 tag for the song that is currently playing, and calls *process_rating()* to create a database entry for the song to store the selected radio button settings.

Installation

You need to install Xmms on your machine to allow Rateplay to work with the MP3 player. Having done so, users can download the Musicus sources from [3], unpack them, and enter *make*. Then you need to copy the *musicus* binary to */usr/bin/*.

The Perl modules *POE*, *PoCo::Player::Musicus*, are *Gtk* are available from CPAN. The article at [2] has a few tips if *Gtk* doesn't install right out of the box. Rateplay also needs the *DBI*, *DBD::SQLite*, *Class::DBI* *Class::DBI::AbstractSearch*, and *Algorithm::Numerical::Shuffle* modules. The CPAN shell should automatically resolve any dependencies that occur.

The Musicus and *POE::Component::Player::Musicus* developers are working hard on enhancing their projects. If the current versions do not work, there are two tar archives at [5] which are guaranteed to work. ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/45/Perl*

[2] Michael Schilli, "Winning Team Player": Linux Magazine 05/04, p. 68

[3] Musicus homepage: *http://muth.org/Robert/Musicus*

[4] SQLite: *http://sqlite.org*

[5] Fallback tarballs for Musicus and PoCo::Player::Musicus: *http://perlmeister.com/musicus*

Listing 1: *rateplay*

```
347         $btn,    1 + $_,
348         2 + $_, 0 + $row,
349         1 + $row
350       );
351     }
352 }
353
354 ##################################
355 sub process_rating {
356 ##################################
357   my $rec =
358     Rateplay::Song
359     ->find_or_create(
360     { path => $SONG } );
361
362   $rec->energize(
363     $RATE_ENERG);
364   $rec->schmoop($RATE_SCHMO);
365   $rec->update();
366 }
367
368 ##################################
369 sub select_songs {
370 ##################################
371   my @energ = grep {
372     $PLAY_ENERG[ $_ - 1 ]
373   } ( 1 .. @PLAY_ENERG );
374
375   my @schmo = grep {
376     $PLAY_SCHMO[ $_ - 1 ]
377   } ( 1 .. @PLAY_SCHMO );
378
379   return sort { rand > 0.5 }
380     map       { $_->path() }
381     Rateplay::Song
382     ->search_where({
383       energize =>
384         [ @energ, 0 ],
385       schmoop =>
386         [ @schmo, 0 ]});
387 }
388
389 ##################################
390 sub next_in_playlist {
391 ##################################
392   my ($backward) = @_;
393
394   return
395     unless scalar @PLAYLIST;
396   my $path;
397
398   {
399     if ($backward) {
400       $path = pop @PLAYLIST;
401       unshift @PLAYLIST,
402               $path;
403     } else {
404       $path =
405         shift @PLAYLIST;
406       push @PLAYLIST, $path;
407     }
408     redo
409       if defined $SONG
410       and $SONG eq $path
411       and @PLAYLIST > 1;
412   }
413
414   $PLAYER->stop();
415   $poe_kernel->post('main',
416     'song', $path);
417 }
418
419 ##################################
420 sub update_rating {
421 ##################################
422   my ($path) = @_;
423
424   if(my ($song) =
425       Rateplay::Song->search(
426         path => $path)) {
427
428     my $e = $song->energize();
429     my $s = $song->schmoop();
430
431     $RATE_SCHMO_BUTTONS[$s-1]
432     ->activate();
433     $RATE_ENERG_BUTTONS[$e-1]
434     ->activate();
435   } else {
436     $RATE_SCHMO_BUTTONS[0]
437       ->activate();
438     $RATE_ENERG_BUTTONS[0]
439       ->activate();
440   }
441 }
442
443 ##################################
444 sub mp3_stdout {
445 ##################################
446   my ($self, $args) =
447     @_[ ARG0 .. $#_ ];
448
449   return
450     if exists
451     $RATED{ $args->{out} };
452
453   push @MP3S, $args->{out};
454
455   $GUI{status}->set(
456     scalar @MP3S . " songs" .
457       " ready for rating.");
458 }
```