rj - abstractions for getting things done

Frank BARKNECHT
50668 Cologne
fbar@footils.org

Abstract

rj is an open source library of Pd abstractions that was developed as part of the RjDj project to support developing reactive music scenes for mobile devices like the iPhone running a variant of Pure Data inside the RjDj player software. As a least common denominator for such environments the object class vocabulary of the core "vanilla" version of Pd is used. This way, the rj library is extremely portable between different distributions of Pd and can be used on many mobile devices with limited hardware, but it is also useful as a general musicians toolbox on common computers.
The rj library strives to provide a minimal, but also fairly complete set of useful tools for musicians including composition helpers, sound generators, input handlers or sound effects. It can be used as a globally installed library but it is also designed to work by being copied into a project folder, thereby making the project or "scene" self-contained and easily distributable.
The rj library has been successfully used by dozens of musicians writing scenes for the RjDj platform or in the popular iPhone app for the movie "Inception".

Keywords

abstraction, library, rjdj, composition, reactive

Introduction

In 2008, Michael Breidenbrücker, one of the co-founders of last.fm, started the project that later became known as RjDj[1]. His vision was a new kind of music played on mobile devices that is not a static track anymore, but creates a dynamic, interactive and algorithmic soundscape able to sense and react to environmental changes.

From the beginning it was clear, that the new player software for this music would have to be able to react in realtime to outside influences. It must be able to express processes and algorithms on a musical metalevel: just to play back pre-recorded tracks or pre-composed patterns would not suffice. Pure Data with its relatively large user base and its liberal BSD license was an obvious candidate as a base for this project. Michael's prior experience as a Max/MSP developer and his friendship with Pd developer Günter Geiger may have played an additional role in the decision to use Pd.

At that time Apple's just introduced iPhone device and AppStore software distribution network similarly was the obvious and only valid target platform, also because Apple's music-savvy user base promised to be a welcoming audience for new listening experiences.

However there also was potential for conflict between Apple's history of offering a "shiny jail" with many restrictions for customers and developers on one hand and the freedom loving Open Source (OS) community that backs Pd's development.

RjDj was started with a series of sprints - laid-back get-togethers of Pd users and developers spanning several days with the goal of exploring ways to marry Pd with the iPhone and to marry iPhone users with Pd composers. One important result of these meetings was the specification of the RjDj Scene file format. "Scene" is the term used to describe a piece of reactive music, it is the analog to the term "Track" in pre-recorded music. A Scene file in RjDj is a zip-compressed directory that includes at least a file called "_main.pd": this is the file that the Pd loader inside of the RjDj app loads and then plays. Further resources can be included anywhere in the zip-archive, notably other abstractions used in _main.pd, soundfiles to play, textfiles with scores and so on. In addition to this, an RDF-file called "index.rdf" holds various metadata like title and composer and a cover image can be included to be displayed in the player.

rj: Goals and restrictions

While not required, the rj library presented here typically also is included in a Scene archive. Being included inside of a music project in this way was the main goal when designing the library.

Technically the rj library is a collection of Pd

[1] http://rjdj.me

abstractions that should support composers of reactive music for mobile devices including iPhone and iPod Touch. It is available on Github at http://github.com/rjdj/rjlib/. Apple's AppStore licensing and developer guidelines create a dangerous minefield for developers who would want to use GPL-licensed code in their project, so it was immediately clear that only a version of Pd stripped of any GPL components could be offered inside of the RjDj app. Any Pd patches that must run inside the RjDj player thus cannot use for example the [expr] objects as these are GPL. The Scene directory can however include additional libraries as long as they are implemented as abstractions: The iOS platform prohibits loading binary externals at runtime.

In the very early versions the rj library was part of the RjDj App. But because updates of the application itself were less frequent than updates to the abstraction library it was removed from the App. Instead it now is typically included in a Scene itself. This way a Scene is self-contained and will continue to function, if the rj library would introduce incompatible changes, because the version included in the Scene still is the one current when the Scene was written.

In fact this approach of bundling all resources for a piece of music into a single directory is not uncommon among composers in general. Some even go so far as to include a version of the current Pd application or externals used and their source code itself into a project directory to archive their composition for future performances. Miller Puckette's "Pd Repertory Project" [2] is an example for this approach, that is characterized by its effort to reduce or remove any dependencies. In the rj library the dependencies are reduced as much as is possible without altering Pd itself. As it turned out the vocabulary offered in Pd vanilla is sufficient to implement most of the functionality commonly required to write reactive music.

Overview of the rj library

The rj library currently contains more than 170 abstractions that somewhat arbitrarily have been arranged into the following topical sections:

- Analysis
- Synths
- Effects
- GUIs
- Mappings
- Controllers
- Utilities

[2] http://crca.ucsd.edu/~msp/pdrp/

Abstractions in each section all start with the lowercase initial letter of the section's heading and an underscore. For example the effect for distortion by reduction of bit depth is called e_bitcrusher.pd.

A special reference abstraction called "OVERVIEW.pd" lists the available objectclasses per section in a format similar to the "help-intro.pd" reference file that opens in Pd if a user chooses to see the "Help" for a canvas background. Contrary to help-intro.pd the rj library's OVERVIEW.pd is empty at first: The list of objects is dynamically created on demand from textfiles. This was implemented to allow users to keep an abstraction instance of [OVERVIEW] in their _main.pd file without encountering a performance loss when the Scene is played.

Library Contents: Analysis

The Analysis section is a place for objectclasses dealing with analysing the environment as sensed by device sensors. Unfortunately it is also the section that still needs the most works, currently only a basic onset detection is included.

Library Contents: Synths

Here you will find object that generate some kind of sound starting from basic, often bandlimited oscillators and sampleplayers up to complete drum machines and a selection of powerful synthesizers designed by Andy Farnell including emulations (or at least approximations) of hardware instruments like the famous Juno by Roland.

Library Contents: Effects

Effects are an important family of objects in RjDj: modifying environmental sounds in realtime as the occur has a profound effect on the listener's perception. This probably is the reason for users to describe their RjDj listening experience as a "digital drug". The rj library includes typical effects like filters, reverb, vocoder, phaser, chorus, pitchshifter, dynamics processor (including compressor, noisegate, limiter and expander) or a granular delay.

Library Contents: GUIs

To its users RjDj is focused on aural experience without an elaborate graphical interface. Still when composing or fine-tuning parameters GUI elements like sliders are immensely useful. The GUI section provides such GUI elements that have been wrapped to offer a simple and consistent visual appearance and work nicely with the parameter control system implemented in many of the other library objects.

Library Contents: Mappings

Mappings translate or convert messages from one domain to another. An example would be a translation of numbers to symbols as in a pitch scale (0 maps to pitch C, 1 maps to C-sharp, 2 maps to D and so on) or a conversion of units (like seconds or samples as units of time). The Mappings sections contains various kinds of mapping related objectclasses.

Library Contents: Controllers

Controllers control other things. The well-known [metro] is such an objectclass: It often is at the center of a whole piece of music. If you stop the metro, everything stops. This section of the library includes some metro-variants like a multimetro, that accepts irregular period lists, or various pattern players. Envelope generators like an ADSR are here as well - they control beginning and end of a note.

This section also is the place for two other C-words: Composition and Conductor. Here you'll find helpers for algorithmic composition like a Markov-chain module, urn and drunk for random numbers or a player for midi files - but these have to be converted to textfiles in advance as there is no midi player in Pd vanilla.

Library Contents: Utilities

This final section holds all the things which do not fit into one of the previous sections. Notably it includes a small selection of the most important list-processing abstractions taken from the [list]-abs library for operations like drip, map, filter and reduce, making it unnecessary in many cases to include a full copy of this useful library in every scene. Here also is the place for the preset saving system based on the SSSAD object by the author and the samplebank system to standardize loading and playback of soundfiles in tables, which we will see in more detail later.

Parameter Control and State Saving

In the next sections several notable features of the rj library will be presented in more depth, starting with the parameter control and preset saving built into (most) rj-abstractions.

The more complex abstractions in the rj library share a common way to control their internal settings including as useful state saving system. In an idea taken from the earlier RRADical/Memento library by the author, the rightmost inlet has a special purpose as a control inlet. It accepts meta-messages to set the state of the object.

An example is the s_fm4 synthesizer: a four operator FM-synthesis abstraction. It has quite a lot of parameters: each operator includes an ADSR-envelope with 4 parameters, a ratio to set the frequency in relation to a base frequency and a volume, so each operator has 6 parameters resulting in a total of 24 parameters. Then a 4x4 modulation matrix can be set, requiring an additional 16 parameters, so there are 40 parameters altogether in the FM4-synthesizer.

All these parameters can be set through the rightmost inlet using meta-messages like "A1 5" to set the Attack value of the first operator's ADSR to 5, or "R3 200" to set release in operator 3 to 200 milliseconds.

Dealing with 40 parameters can result in messy patches with lots of patchcords going over each other, unless one takes some smart countermeasures. In the rj library a special abstraction called [u_dispatch] helps with clean patching.

Its main purpose is to "dispatch" meta-messages to receivers. Every [u_dispatch] object has to be created with two arguments that are concatenated with a dash "-" to the receiver's name. The second argument specifies the selector of the meta-message that the dispatcher should handle. The first argument usually is simply $0, to localize the receiver. Internally a [route $2] object will filter out incoming messages. If it finds a matching meta-message it will send it to a receiver called [s $1-$2]. Other messages will go to the outlet of the dispatcher.

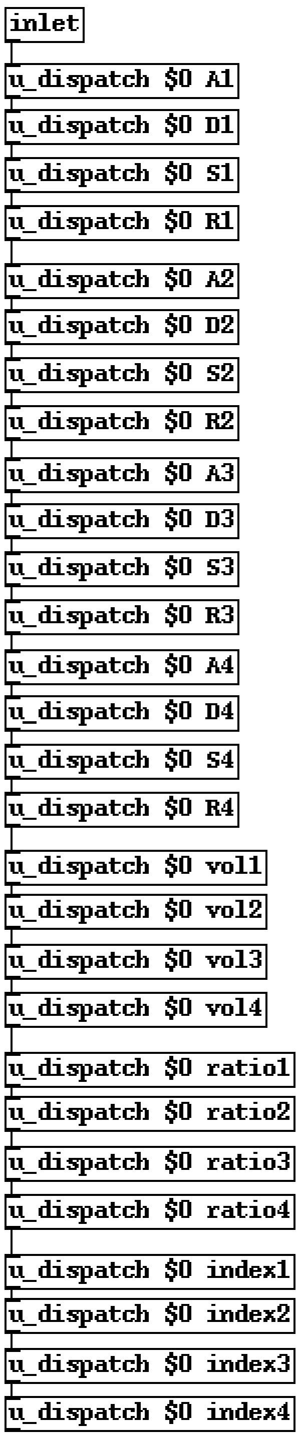


Figure 1: Typical clean parameter-dispatching in rj-abstractions

Taking the examples from above, we would need two dispatchers to handle A1- and R3-meta-messages, that would be created as such (also see figure 1):

```
[u_dispatch $0 A1]
|
[u_dispatch $0 R3]
```

Both are connected to each other and the first one is connected to the control inlet. The attack message from above, "A1 5", will be filtered out in the first dispatcher resulting in a float 5 being sent to the receiver "$0-A1".

The message "R3 200" however will pass through the first dispatcher unchanged, until the second dispatcher will send it to "$0-R3".

[u_dispatch] also handles lists as values in meta-messages. This is used to set the modulation matrix per operator as a four-value list. The meta-message to set the modulation indices for the first operator could look like this: "index1 0 0.05 0 0".

As a full example, here is the parameter list describing a setup sounding similar to the Rhodey electric piano as is part of the STK C++-library:

```
A1 1
A2 1
A3 1
A4 1

D1 1500
D2 250
D3 1500
D4 1000

R1 40
R2 40
R3 40
R4 40

S1 0
S2 0
S3 0
S4 0

index1 0 0 0 0
index2 0 0 0 0
index3 0.5 0 0 0
index4 0 0.1 0 0

ratio1 1
ratio2 1
ratio3 0.5
ratio4 15

vol1 1
vol2 1
vol3 0
vol4 0
```

Abstractions using the dispatcher mechanism can also report their current state, because a second feature of the u_dispatch objects is to provide a link to a parameter saving system based on the SSSAD system by the author. A copy of the sssad.pd abstraction that makes up the system is included in the utilities section under the name u_sssad.pd. Every u_dispatch object includes an instance of [u_sssad] to store the current parameter value into a SSSAD-slot named after the parameter name.

To activate the SSSAD-based saving, an abstraction also needs to include an instance of the u_loader abstraction. It serves as a bridge between the state as stored locally and a global receiver system. Optionally the u_loader's outlet can be connected to the abstraction's control outlet (the rightmost one) and report the state there as well.

To be able to differentiate between various instances of the same abstraction, every occurrence has to be "tagged" with a symbol as first argument. This tag is transported into the preset system via the first argument of the loader, which then by convention should be created as [u_loader <name_of_abstraction>-$1 $0]. The second argument, usually $0, selects the dispatchers to talk to. Remember that these have $0 as their first argument.

When this construction is complete, a full state as presented above can be reported through a set of global senders and receiver. Sending a "save" message to [s RJ_SCENE_SAVE] will make all parameters held in some u_dispatch-object report their current state to a receiver called [r RJ_SCENE] in a format, that includes the specific tags of abstractions with the abstraction name prepended. This also is the format used for restoring or remote-control of settings via the global receiver "RJ_SCENE_LOAD".

Again an example should illustrate this: Assuming an instance of the FM4-synthesizer tagged like this: [s_fm4 MYTAG]. On "save" to RJ_SCENE_SAVE, the following may be received on [r RJ_SCENE]:

```
s_fm4-MYTAG A1 1,
s_fm4-MYTAG A2 1,
s_fm4-MYTAG A3 1,
s_fm4-MYTAG A4 1,
s_fm4-MYTAG D1 1500,
s_fm4-MYTAG D2 250,
s_fm4-MYTAG D3 1500,
s_fm4-MYTAG D4 1000,
...
```

Note that the order is undefined! These messages could then be saved into a [textfile] object or to a message box object. To restore the parameters, the messages simply can be sent to [s RJ_SCENE_LOAD]. Alternatively they can be stripped of their first element ("s_fm4-MYTAG" here) to feed them directly into the control inlet. Figure 2 shows a complete patch saving the state of a dispatcher-enabled abstraction into a message box. The message box can be activated or "loadbanged" to restore settings. Saving into a textfile-object would follow the same principle.

Sample Management

Playing back prerecorded samples or recordings that have been taken in realtime as snapshots of environmental sounds is a common task in many RjDj-Scenes. Several scenes produced by the team at Reality Jockey and other producer also are "reactive remixes" of tracks by major artists like the French duo "Air". Here traditionally recorded stems have been used and enhanced with reactive elements.

The rj library supports these tasks with several utility objects that together form an architecture for sample management in tables. (Playback from disk is delegated to the usual [readsf~] object.)

At the core of rj's soundfile management is a task sharing concept: Several objects deal with loading of soundfiles into tables while other objects play back these tables in response to commands carrying the table identifiers.

The basic file-loader is the [u_samplebank] object. It is a rather thin wrapper around a [soundfiler] object and a target table, whose name has to be supplied as first argument. The soundfile to load into the table can be supplied as second argument or via a message to the samplebank's inlet. The samplebank reports the current table name and length in samples and milliseconds and the samplerate used to calculate it (default is 22050 Hz, which can be overridden) to its outlet in response to an "info" message received on the inlet.

The corresponding playback object is called [s_playtable]. It is modelled after a similar object in Karlheinz Essl's RTC-library [3] and will play back a samplebank-table when it receives the samplebank's table-identifier name, optionally transposing or attenuating it or playing only a part of

[3] http://www.essl.at/works/rtc.html

the table.

Both [u_samplebank] and [s_playtable] work with mono files - stereo samples are handled by [u_samplebank2] and [s_playtable2]. Because stereo files require two tables for left and right channels, the argument specifying the table in the samplebank now is not the actual table name but a base name for the two tables inside. Their real names have "-1" and -2" appended. The stereo playback object expects just the basename in its activation command and will automatically extend the names following the same pattern internally.

Several extensions to the basic samplebank objects support loading multiple files into tables or even a directory of samples with the [u_samplekit] objects.

Live recordings are simplified with the [u_record] object: It is very similar to the samplebank, but instead of opening a file it expects a "record" meta-message to start recording. In addition to the table-name a buffer length to resize the internal table is required. The "info" meta-message will report the length of the last recording.

To an experienced Pd user these objects may look simple, and indeed inside they don't provide much advanced functionality beyond what is usually patched around a [soundfiler]. But it turned out that musicians coming from a more traditional background were happy to find familiar terms like "samplebank" or "samplekit" instead of "soundfiler" or "record" instead of "tabwrite~". And even Pd power-users can get bored of patching the same "bang -> openpanel -> "read -resize $1" -> soundfiler" again and again and will find it useful to get the table length with a simple "info" command even long after a file was loaded (which is the only time a naked [soundfiler] will report the samples loaded).

Powerful Synthesizers

The original target platform for RjDj was the first generation iPhone. While it was very fast for a mobile device at the time, it still was necessary to be careful with CPU-intensive processes. This is one reason for the "half-CD" samplerate of 22050 Hz used in the RjDj app. Over time mobile phones including the iPhone became faster, so in 2010 it became feasible to add some powerful synthesizers to the rj library, which would max out the iPhone 1G, but run very well on the iPhone 4. These synthesizer were written by Andy Farnell, author of a book on realtime sounddesign called "Designing Sound". They approximate several well-known hardware synthesizer models from the past. Andy not only wrote the synthesizer patches, he also provided a collection of preset sounds for each abstraction, that can serve as a base for further exploration and showcase the capabilities of the synth-abstractions.

Designing preset sounds is a time-consuming task where a lot of "fiddling" and fine tuning is necessary. To make this process easier, every synth comes with a graphical user interface providing quick access to the internal parameters. All synths share the same basic interface structure, only parameter names are different and controls might be disabled for certain synths. Figure 3 shows the programmer for the s_ejun-synthesizer. Note the greyed-out controls for two Oscillators and the custom names for several modifiers, which would be different in other synths.

For saving and restoring presets, the parameter handling system described above is used. The GUI itself uses abstractions of the GUI-section in the rj-library, that automatically prepend a tag in front of their values. The sliders in the screen shot thus not only produce a number, but a meta-message, for example "mod1a 0.7". The meta-message-names of all synths are unified, so the same GUI patch can be reused. The ranges are normalized to be 0 to 1. The programmer GUI itself is not part of the synthesizer abstraction, in fact, it only is part of the help-file.

Future work

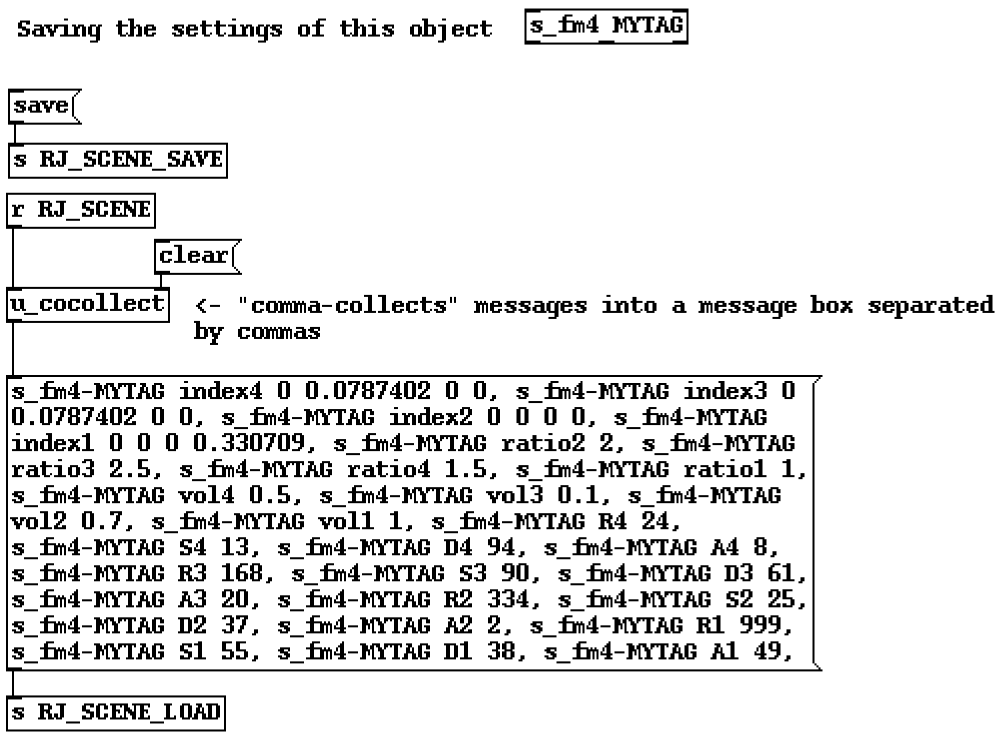
When developing any library deciding which objectclasses to include and which should be left out maybe is the most challenging duty of the maintainer. A guideline in solving this for the rj library always was to provide a minimal and small library, that still is sufficient 80 to 90 percent of the time. For example, while it would have been possible to include the full [list]-abs collection to provide a complete set of list-processing operations, only the half a dozen most important objects from there have been included. I estimate that they fulfil most list-demands composers have. For the remaining more advanced tasks the composers could just copy the wanted abstractions from the original library.

The analysis section obviously needs some more work, but this is underway. However even as it stands, the rj library has proven to be a useful resource not only for writing RjDj scenes. In fact, the author also uses it in most of his outside projects. It's easy to "install" by copying it into a project directory and saves a lot of headaches in regard to outside dependencies. This last point makes it especially useful in a teaching context as well. Not only can beginning users avoid having to hunt down exotic externals, they also can take a look inside the rj abstractions to see how certain things can be implemented in pure Pd, and directly copy it into their own variations.

All in all rj has proven to be a nice library for "just getting things done" in a fun way.

Acknowledgements

Our thanks go to everyone at Reality Jockey for supporting development of the library and testing, testing, testing it in real world applications, particularly Joe White, Robert Thomas, Florian Waldner and Andy Farnell. Special thanks go to the authors of abstractions that have found their way into the rj library in pure or slightly modified form. Among these are Miller Puckette, Roman Haefeli, Günter Geiger, Amaury Hazan and Mathieu Bouchard.



```
Saving the settings of this object  [s_fm4 MYTAG]

[save(
|
[s RJ_SCENE_SAVE]

[r RJ_SCENE]
|
|          [clear(
|          |
[u_cocollect]  <- "comma-collects" messages into a message box separated
               by commas
|
[s_fm4-MYTAG index4 0 0.0787402 0 0, s_fm4-MYTAG index3 0
0.0787402 0 0, s_fm4-MYTAG index2 0 0 0 0, s_fm4-MYTAG
index1 0 0 0 0.330709, s_fm4-MYTAG ratio2 2, s_fm4-MYTAG
ratio3 2.5, s_fm4-MYTAG ratio4 1.5, s_fm4-MYTAG ratio1 1,
s_fm4-MYTAG vol4 0.5, s_fm4-MYTAG vol3 0.1, s_fm4-MYTAG
vol2 0.7, s_fm4-MYTAG vol1 1, s_fm4-MYTAG R4 24,
s_fm4-MYTAG S4 13, s_fm4-MYTAG D4 94, s_fm4-MYTAG A4 8,
s_fm4-MYTAG R3 168, s_fm4-MYTAG S3 90, s_fm4-MYTAG D3 61,
s_fm4-MYTAG A3 20, s_fm4-MYTAG R2 334, s_fm4-MYTAG S2 25,
s_fm4-MYTAG D2 37, s_fm4-MYTAG A2 2, s_fm4-MYTAG R1 999,
s_fm4-MYTAG S1 55, s_fm4-MYTAG D1 38, s_fm4-MYTAG A1 49,(
|
[s RJ_SCENE_LOAD]
```

Figure 2: Saving parameters of rj-objects into a message box

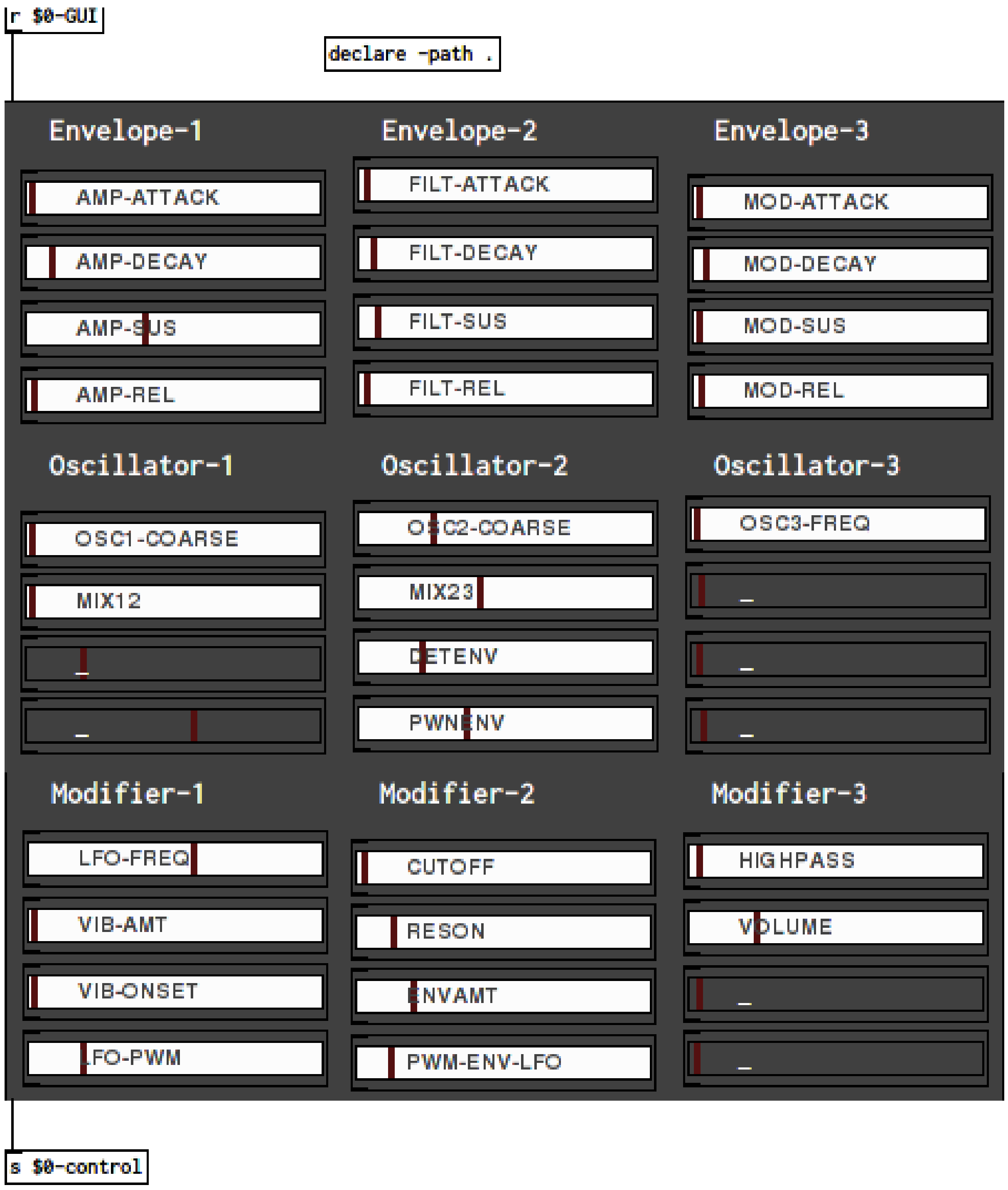


Figure 3: Programmer for the s_ejun synthesizer