RRADical Pd

Author: Frank Barknecht <fbar@footils.org>

Abstract

RRADical Pd is a project to create a collection of Pd patches, that make Pd easier and faster to use for people who are more comfortable with commercial software like Reason(tm) or Reaktor(tm). RRAD as an acronym stands for "Reusable and Rapid Audio Development" or "Reusable and Rapid Application Development", if it includes non-audio patches, with Pd. In the design of this system, a way to save state flexibly in Pd (persistence) had to be developed. For communication among each other the RRADical patches integrate the Open Sound Control protocol.

What it takes to be a RRADical

RRAD as an acronym stands for "Reusable and Rapid Audio Development" or "Reusable and Rapid Application Development", if it includes non-audio patches, with Pd. It is spelled RRAD, but pronounced "Rradical" with a long rolling "R".

The goal of RRADical Pd is to create a collection of patches, that make Pd easier and faster to use for people who are more used to software like Reason(tm) or Reaktor(tm). For that I would like to create patches, that solve real-world problems on a higher level of abstraction than the standard Pd objects do. Where suitable these high level abstractions should have a graphical user interface (GUI) built in. As I am focused on sound production the currently available RRADical patches mirror my preferences and mainly deal with audio, although the basic concepts would apply for graphics and video work using for example the Gem and PDP extensions as well.

Pre-fabricated high-level abstractions may not only make Pd easier to use for beginners, they also can spare lot of tedious, repeating patching work. For example building a filter using the `lop~` object of Pd usually involves some way of changing the cutoff frequency of the filter. So another object, maybe a slider, will have to be created and connected to the `lop~`. The typing and connecting work has to be done almost every time a filter is used. But the connections between the filter's cutoff control and the filter can also be done in advance inside of a so called abstraction, that is, in a saved Pd patch file. Thanks to the Graph-On-Parent feature of Pd the cutoff slider even can be made visible when using that abstraction in another patch. The new filter abstraction now carries its own GUI and is immediately ready to be used.

Of course the GUI-filter is a rather simple example (although already quite useful). But building a graphical note sequencer with 32 sliders and 32 number boxes or even more is something, one would rather have to do only once, and then reuse in a lot of patches.

Problems and Solutions

To build above, highly modularized system several problems have to be solved. Two key areas turned out to be very important:

Persistence How to save the current state of a patch? How to save more than one state (state sequencing)?

Communication The various modules are building blocks for a larger application. How should they talk to each other. (In Reason this is done by patching the back or modules with horrible looking cables. We must do better.)

It turned out, that both tasks are possible to solve in a consistent way using a unique abstraction. But first lets look a bit deeper at the problems at hand.

Persistence

Pd offers no direct way to store the current state of a patch. Here's what Pd author Miller S. Puckette writes about this in the Pd manual in section "2.6.2. persistence of data":

> Among the design principles of Pd is that patches should be printable, in the sense that the appearance of a patch should fully determine its functionality. For this reason, if messages received by an object change its action, since the changes aren't reflected in the object's appearance, they are not saved as part of the file which specifies the patch and will be forgotten when the patch is reloaded.

Still, in a musician's practice some kind of persistence turns out to be an important feature, that many Pd beginners do miss. And as soon as a patch starts to use lots of graphical control objects, users will - and should - play around with different settings until they find some combination they like. But unless a way to save this combination for later use is found, all this is temporary and gone, as soon as the patch is closed.

There are several approaches to add persistence. Max/MSP has the `preset`-object, Pd provides the similar `state`-object which saves the current state of (some) GUI objects inside a patch. Both objects also support changing between several different states.

But both also have at least two problems: They only save the state of GUI objects, which might not be everything that a user wants to save. And they don't handle abstractions very well, which are crucial when creating modularized patches.

Another approach is to (ab)use some of the Pd objects that can persist itself to a file, especially `textfile`, `qlist` and `table`, which works better, but isn't standardized.

A rather new candidate for state saving is Thomas Grill's `pool` external. Basically it offers something, that is standard in many programming languages: a data structure that stores key-value-pairs. This structure also is known as hash, dictionary or map. With `pool` those pairs also can be stored in hierarchies and they can be saved to or loaded from disk. The last but maybe most important feature for us is, that several pools can be shared by giving them the same name. A `pool MYPOOL` in one patch will contain the same data as a `pool MYPOOL` in another patch. Changes to one pool will change the data in the other as well. This allows us to use `pool MYPOOL`s inside of abstractions, and still access the pool from modules outside the abstractions, for example for saving the `pool` to disk.

A `pool` object is central to the persistence in RRADical patches, but it is hidden behind an abstracted "API", if one could name it that. I'll come back to how this is done below.

Communication

Besides persistence it also is important to create a common path through which the RRADical modules will talk to each other. Generally the modules will have to use, what Pd offers them, and that is either a direct connection through patch cords or the indirect use of the send/receive mechanism in Pd. Patch cords are fine, but tend to clutter the interface. Sends and receives on the other hand will have to make sure, that no name clashes occur. A name clash is, when one target receives messages not intended for it. A patch author has to remember all used send-names, which might be possible, if he did write the whole patch himself and kept track of the send-names used. But this gets harder to impossible, if he uses prefabricated modules, which might use their own senders, maybe hidden deep inside of the module.

So it is crucial, that senders in RRADical abstractions use local names only with as few exceptions as possible. This is achieved by prepending the RRADical senders with the string "$0-". So instead of a sender named `send volume`, instead one called `send $0-volume` is used. $0 makes those sends local inside their own patch borders by being replaced with a number unique to that patch. Using $0 that way is a pretty standard idiom in the Pd world.

Still we will want to control a lot of parameters and do so not only through the GUI elements Pd offers, but probably also through other ways, for example through hardware Midi controllers, through some kind of score on disk, through satellite navigation receivers or whatever.

This creates a fundamental conflict:

We want borders We want to separate our abstraction so they don't conflict with each other.

We want border crossings We want to have a way to reach their many internals and control them from the outside.

The RRADical approach solves both requirements in that it enforces a strict border around abstractions but drills a single hole in it: the **OSC inlet**. This idea is the result of a discussion on the Pd mailing list and goes back to suggestions by Eric Skogen and Ben Bogart. Every RRADical patch has (to have) a rightmost inlet that accepts messages formatted according to the OSC protocol. OSC stands for Open Sound Control and is a network transparent system to control (audio) applications remotely and is developed at CNMAT in Berkley by Matt Wright mainly.

The nice thing about OSC is that it can control many parameters over a single communication path (like a network conneciton using a definite port). For this OSC uses a URL-like scheme to address parameters organized in a tree. An example would be this message:

```
/synth/fm/volume 85
```

It sends the message "85" to the "volume" control of a "fm" module below a "synth" module. OSC allows many parameters constructs like:

```
/synth/fm/basenote              52
/synth/virtualanalog/basenote   40
/synth/*/playchords             m7b5 M6 7b9
```

This might set the base note of two synths, fm and virtualanalog and send a chord progression to be played by both – indicated by the wildcard * – afterwards.

The OSC-inlet of every RRADical patch is intended as the border crossing: Everything the author of a certain patch intends to be controlled from the outside can be controlled by OSC messages to the OSC-inlet. The OSC-inlet is strongly recommended to be the rightmost inlet of an abstraction. At least all of my RRADical patches do it this way.

Trying to remember it all: Memento

To realize the functionality requirements laid out so far I resorted to a so called Memento. "Memento" is a very cool movie by director Christopher Nolan where - quoting IMDB:

> A man, suffering from short-term memory loss, uses notes and tattoos to hunt down his wife's killer.

The movie's main character Leonard has a similar problem as Pd: he cannot remember things. To deal with his persistence problem, his inability to save data to his internal harddisk (brain) he resorts to taking a lot of photos. These pictures act as what is called a Memento: a recording of the current state of things.

In software development Mementos are quite common as well. The computer science literature describes them in great detail, for example in the Gang-Of-Four book "Design Patterns" [Gamma95]. To make the best use of a Memento science recommends an approach where certain tasks are in the responsibility of certain independent players.

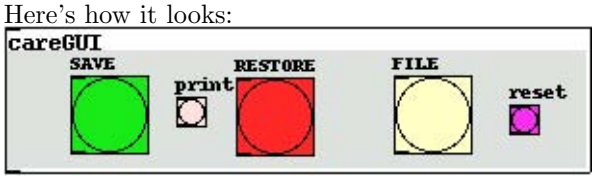
The Memento itself, as we have seen, is the photo, i.e. some kind of state record. A module called the "Originator" is responsible for creating this state and managing changes in it. In the movie, Leonard is the Originator, he is the one taking photos of the world he is soon to forget.

The actual persistence, that could be the saving of a state to harddisk, but could just as well be an upload to a webserver or a CVS check-in, is done by someone called the "Caretaker" in the literature. A Caretaker could be a safe, where Leonard puts his photos, or could be a person, to whom Leonard gives his photos. In the movie Leonard also makes "hard saves" by tattooing himself with notes he took. In that case, he is not only the Originator of the notes, but also the Caretaker in one single person. The Caretaker only has to take care, that those photos, the Mementos, are in a safe place and no one fiddles around with them. Btw: In the movie some interesting problems with Caretakers, who don't always act responsible, occur.

Memento in Pd

I developed a set of abstractions, of patches for Pd, that follow this design pattern. Memento for Pd includes a `caretaker` and an `originator` abstraction, plus a third one called `commun` which is responsible for the **internal** communication. `commun` basically is just a thin extension of `originator` and should be considered part of it. There is another patch, the `careGUI` which I personally use instead of the `caretaker` directly, because it has a simple GUI included.

Here's how it looks:



[Gamma95] E. Gamma and R. Helm and R. Johnson and J. Vlissides: "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley 1995

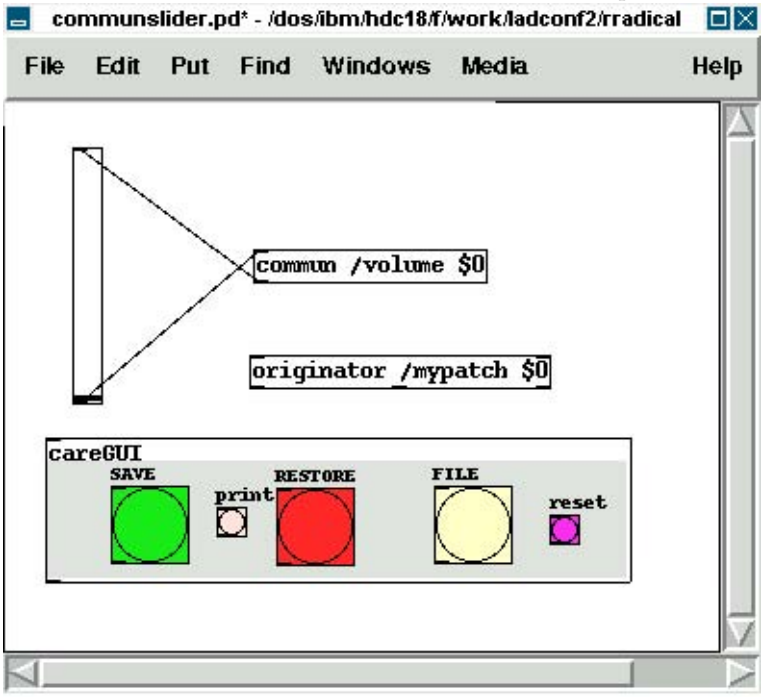
The careGUI is very simple: select a FILE-name to save to, then clicking SAVE you can save the current state, with RESTORE you can restore a state previously saved. After restore, the outlet of careGUI sends a bang message to be used as you like.

Internally caretaker has a named pool object using the global pool called "RRADICAL". The same pool RRADICAL also is used inside the originator object. This abstraction handles all access to this pool. A user should not read or write the contents of pool RRADICAL directly. The originator patch also handles the border crossing through OSC messages by its rightmost inlet. The patch accepts two mandatory arguments: The first on is the name under which this patch is to be stored inside the pool data. Each originator SomeName secondarg stores it's data in a virtual subdirectory inside the RRADICAL-pool called like its first argument - SomeName in the example. If the SomeName starts with a slash like "/patch" , you can also access it via OSC through the rightmost inlet of originator under the tree "/patch"

The second argument practically always will be $0. It is used to talk to those commun objects which share the same second argument. As $0 is a value local and unique to a patch (or to an abstraction to be correct) each originator then only can talk to communs inside the same patch and will not disturb other commun objects in other abstractions.

The commun objects finally are where the contents of a state are read and set. They, too, accept two arguments, the second of which was discussed before and will most of the time just be $0. The first argument will be the key under which some value will be saved. You should use a slash as first character here as well to allow OSC control. So an example for a usage would be commun /vol $0.

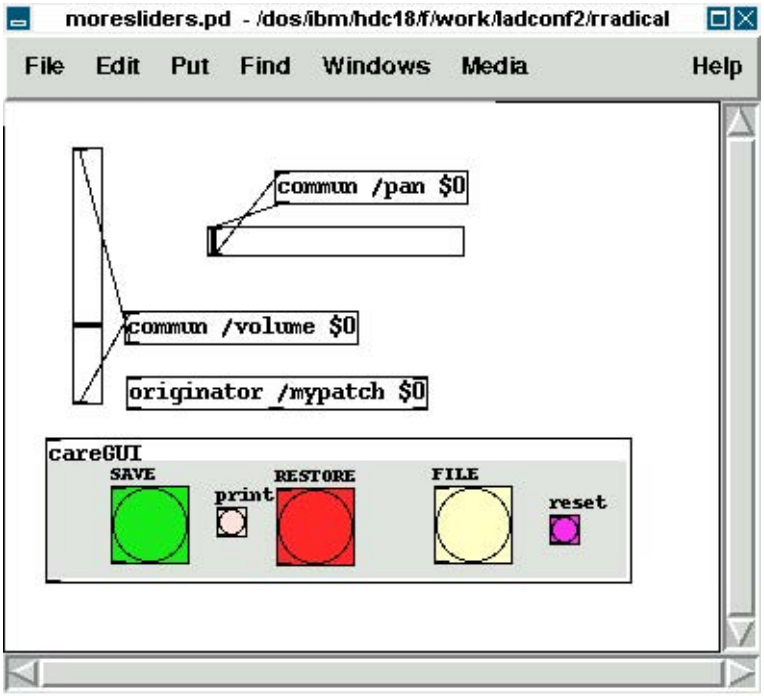
commun has one inlet and one outlet. What comes in through the inlet is send to originator who stores it inside its Memento under the key, that is specified by the commun's first arg. Actually originator. The outlet of a commun will spit out the current value stored under its key inside the Memento, when originator tells it to do so. So communs are intended to be cross-connected to some thing that can change. And example would be a slider which can be connected as seen in the next picture:



In this patch, every change to the slider will be reflected inside the Memento. The little print button in careGUI can be used to print the contents to the console from which Pd was started. Setting the slider will result in something like this:

```
/mypatch 0 , /volume , 38
```

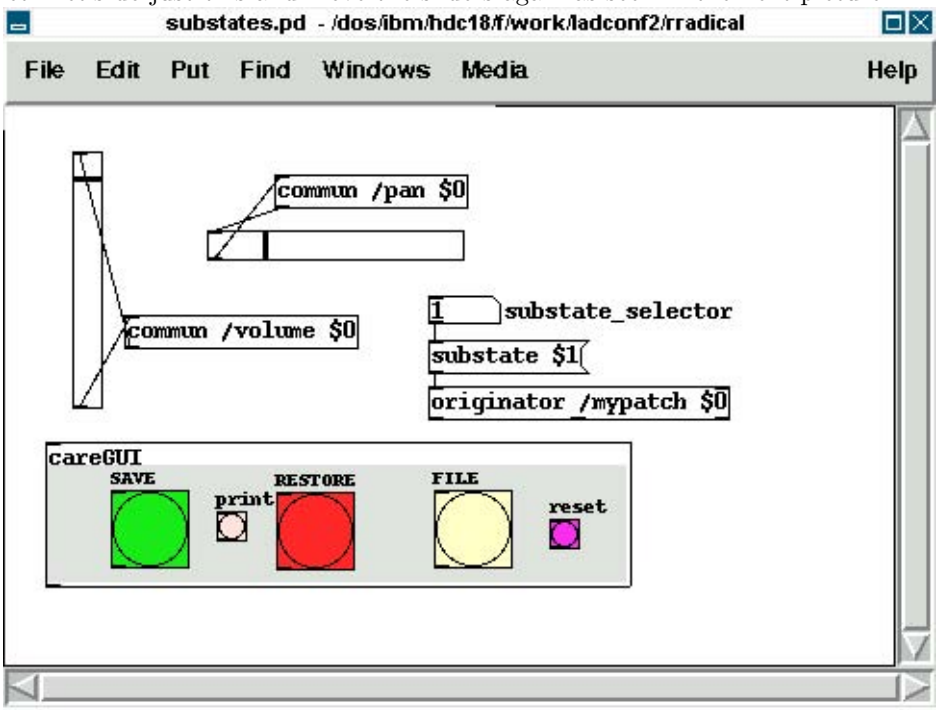
Here a comma separates key and value pairs. "mypatch" is the top-level directory. This contains a 0, which is the default subdirectory, after that comes the key "/volume", whose value is 38. Let's add another slider for pan-values:



Moving the /pan slider will let careGUI print out:

```
/mypatch 0 , /volume , 38
/mypatch 0 , /pan , 92
```

The `originator` can save several substates or presets by sending a `substate #number` message to its first inlet. Let's do just this and move the sliders again as seen in the next picture:



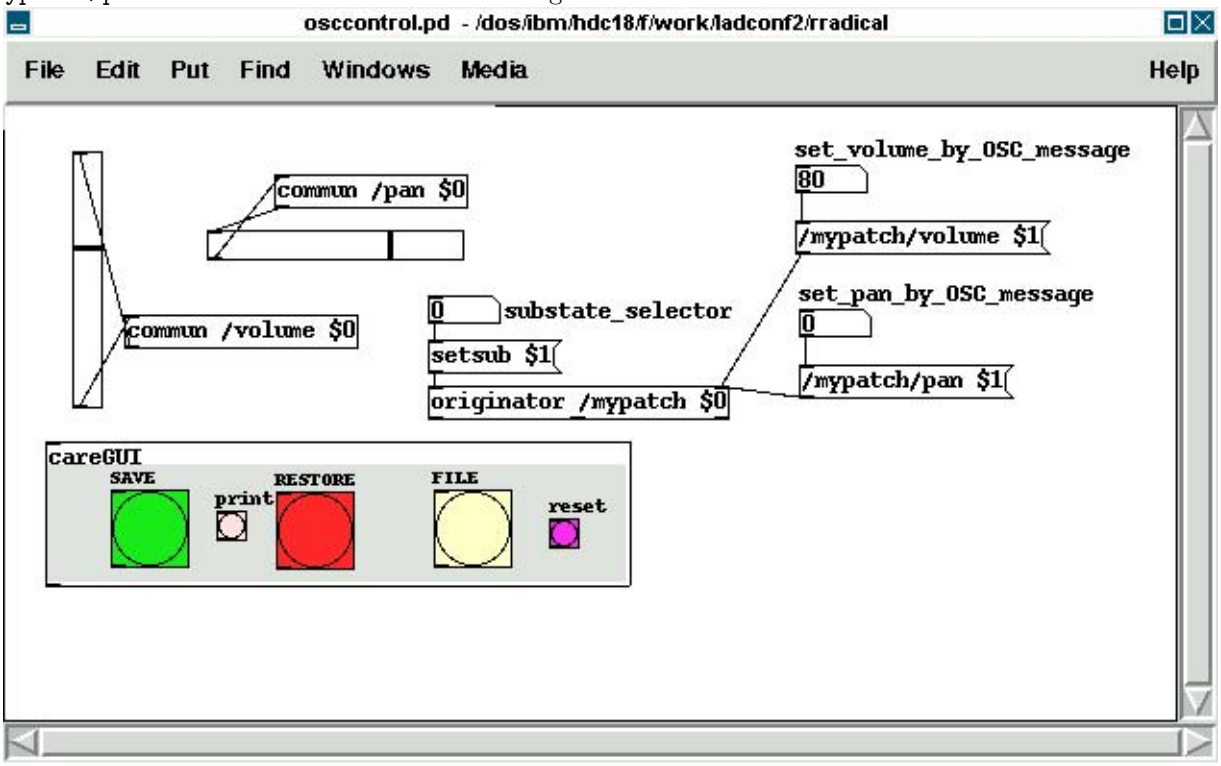
Now careGUI prints:

```
/mypatch 0 , /volume , 38
/mypatch 0 , /pan , 92
/mypatch 1 , /volume , 116
/mypatch 1 , /pan , 27
```

You see, the substate 0 is unaffected, the new state can have different values. Exchanging the `substate` message with a `setsub` message will autoload the selected state and "set" the sliders to the stored values immediately.

OSC in Memento

The whole system now already is prepared to be used over OSC. You probably already guess, how the message looks like. Any takers? Thank you, you're right, the messages are built as `/mypatch/volume #number` and `/mypatch/pan #number` as shown in the next stage:



Sometimes it is useful to also get OSC messages out of a patch, for example to control other OSC software through Pd. For this the **OSC-outlet** of `originator` can be used, which is the rightmost outlet of the abstraction. It will print out every change to the current state. Connecting a `print OSC` debug object to it, we get to see what's coming out of the OSC-outlet when we move a slider:

```
OSC: /mypatch/pan 92
OSC: /mypatch/pan 91
OSC: /mypatch/pan 90
OSC: /mypatch/pan 89
```

Putting it all to RRADical use

Now that the foundation for a general preset and communication system are set, it is possible to build real patches with it that have two main characteristics:

Rapidity Ready-to-use high-level abstraction can save a lot of time when building larger patches. Clear communication paths will let you think faster and more about the really important things.

Reusability Don't reinvent the wheel all the time. Reuse patches like instruments for more than one piece by just exchanging the Caretaker-file used.

I already developed a growing number of patches that follow the RRADical paradigm, among these are a complex pattern sequencer, some synths and effects and more. All those are available in the Pure Data CVS, which currently lives at pure-data.sourceforge.net in the directory "abstractions/rradical". The RRADical collection comes with a template file, called `rrad.tpl.pd` that makes deploying new RRADical patches easier and lets developers concentrate on the algorithm instead of bookkeeping. Some utilities help with creating the sometimes needed many `commun`-objects. Several usecases show example applications of the provided abstractions.

Much, but not all is well yet

Developing patches using the Memento system and the design guidelines presented has made quite an impact on how my patches are designed. Before Memento quite a bit of my patches' content dealed with saving state

in various, crude and non-unified ways. I even tried to avoid saving states at all because it always seemed to be too complicated to bother with it. This limited my patches to being used in improvisational pieces without the possibility to prepare parts of a musical story in advance and to "design" those pieces. It was like being forced to write a book without having access to a sheet of paper (or a harddisk nowadays). This has changed: having "paper" in great supply now has made it possible to "write" pieces of art, to "remember" what was good and what rather should not be repeated, to really "work" on a certain project over a longer time.

RRADical patches also have proven to be useful tools in teaching Pure Data, which is important as usage of Pd in workshops and at universities is growing – also thanks to its availability as Free Software. RRADical patches directly can be used by novices as they are created just like any other patch, but they already provide sound creation and GUI elements that the students can use immediately to create more satisfactory sounds that the sine waves used as standard examples in basic Pd tutorials. With a grown proficiency the students later can dive into the internals of a RRADical patch to see what's inside and how it was done. This allows a new top-down approach in teaching Pd which is a great complement (or even alternative) to the traditional, bottom-up way.

Still the patches suffer from a known technical problem of Pd. Several of the RRADical patches make heavy use of graphical modules like sliders or number boxes, and they create a rather high number of messages to be send inside of Pd. The message count is alleviated a bit by using OSC, but the graphical load is so high, that Pd's audio computation can be disturbed, if too many GUI modules need updating at the same time. This can lead to dropouts and clicks in the audio stream, which is of course not acceptable.

The problem is due to the non-sufficient decoupling of audio and graphics rsp. message computations in Pd, a technical issue that is known, but a solution to my knowledge could require a lot of changes to Pd's core system. Several developers already are working on this problem, though.

The consistent usage of OSC throughout the RRADical patches created another interesting possibility, that of collaboration. As every RRADcial patch not only can be controlled through OSC, but also can control another patch of its own kind, the same patch could be used on two or more machines, and every change on one machine would propagate to all other machines where that same patch is running. So jamming together and even the concept of a "Pd band" is naturally build into every RRADcial patch.