Let's Make Libraries!

Forging a common path through the glorious forkiness of Pd

Hans-Christoph Steiner
Interactive Telecommunications Program
New York University
hans@at.or.at

"Namespaces are one honking great idea – let's do more of those!" – The Zen of Python[1]

ABSTRACT

Pd is more of an common set of concepts rather than a single software package. There are many branches, forks and packages of Pd, and Pd itself began life as a sort of fork from IRCAM's Max. Rather than viewing this as a problem, it is instead part of the strength of the Pd community. In order to support this nature, a common library format is needed so that code can be reused and shared between all of these versions. This paper reviews how other programming environments handle the same set of problems, then summarizes the years of discussion on this topic within the Pd community, and finally proposes a common library format.

1. INTRODUCTION

Pd has a long history of branches, forks and multiple distros[1], and indeed Pd itself started as a conceptual fork of Max, which also has a history of forks and branches. Rather than view this as a weakness, it is instead a strength of the code and the Pd community. Pd has been developed by people who use it as a primary tool in their own creations. When people take ownership of their tools, they want to tweak and customize them. This is not only a natural impulse, it is a good one: it allows for unfettered creativity and allows new ideas and ways of working to develop. The downsides come with incompatibilities and problems managing the vast array of libraries and objects that are available for Pd.

While most programming languages are thought of as a single unified entity, there are nonetheless lessons that the Pd community can learn from to address the issues surrounding reusing and distributing libraries of code. In particular, Tcl, Python, and Lua have share some similarities in spirit with Pd, and all have addressed the issues of namespaces and libraries. The central goal is that a Pd patch written on one computer with a certain distro should then work on any distro, as long as it includes the needed libraries. Pd can learn from Lua, Python and Tcl to create a common library format, namespaces, and the basic syntax for managing them.

[1] `http://www.python.org/dev/peps/pep-0020/`
[1] refers to a distribution of software

Permission to make digital or hard copies of all or part of this work for any purpose are granted under a Free Art License 1.3: `http://artlibre.org/licence/lal/en/`
The Third International Pure Data Convention, São Paolo, Brazil
Copyright 2009 Copyright remains with the author(s).

2. PROBLEMS

2.1 Multi-object-single-file Name Clashes

Pd has been able to load libraries for a long time, and there were initially two library formats: multiple object-classes linked into a single file (multi-object-single-file) or collections of files where each file represents a single object-class. These collections of single files could be made up of both binaries and .pd files. Initially, most libraries of binaries were distributed in this multi-object-single-file format. The way that Pd is currently implemented, using multi-object-single-file libraries makes it impossible to use an objectclass if one with the same name had already been loaded. This causes problems in situations like *Gem*'s [`scale`] vs. *maxlib*'s [`scale`]. Both are using the word "scale" quite logically and appropriately, but for different functionalities. If one is loaded before the other, the other is completely unusable only using the name [`scale`]. This was addressed in Pd-extended by distributing as many libraries as possible using single-class-single-file collections organized into *libdirs* [11]. This allowed objects like *maxlib*'s [`scale`] to be accessed with a namespace prefix, e.g. [`maxlib/scale`].

2.2 Names and Communication

For many years, it has been suggested that we really should be solving name clashes by increasing communication. Increased communication is generally beneficial in a community, and can be quite useful in solving issues, provided that the community is not too large to allow for effective communication and dissemination of ideas. In this particular set of problems, we should ask people to not use names that already exist, and with Pdpedia and other tools, this is easier to do. Also, names should be descriptive and distinct, and that will also decrease the likelihood of name clashes.

Increased communication should always be encouraged when there are issues within a community, but there are many factors which mean it is not feasible for these issues to be addressed by communication alone. The first is the question of how many Pd communities there are. The Pd

lists hosted by IEM[2] have long been the central meeting place of the core developers, but there are many other forums as well. People in these forums are generating useful code that is often shared. If people are not part of the IEM lists, it makes it more difficult to find and communicate with them. Also, speaking English in the English forums should not be a requirement for producing reusable libraries.

And lastly, if people do not participate in the main forums and follow the rules devised there, they might still be producing useful code. If they haven't followed the communication for avoiding problems, they might have unknowingly broken these rules. So it would be difficult to use that code if there was a name clash. If we provide a set of tools that make it easy to avoid the name clashes and other related issues, then we have a solid foot to stand on when discussion among developers is not effective. In order for these tools to gain widespread use, they need to be simple, intuitive, and well-documented.

2.3 Loading all the libraries

With Pd-extended another approach tried to combine the ease of a global namespace with a solution to access specific objectclasses in the case of name collisions. The libraries were separated into *libdirs* then all of them were loaded into the global namespace by default. This provided the ease of use of the technique of putting all the objectclasses into a global namespace, but still allowed specific objectclasses to be loaded with a namespace prefix. This approach also has proved to have fundamental flaws. For example, take two libraries, *firstlib* and *secondlib*. *firstlib* is loaded first, *secondlib* is loaded second. *secondlib* has an object called [foo]. A lot of people use it frequently. It is included in lots of patches just as [foo] since *firstlib* is loaded into the global namespace. The author of *firstlib* never uses *secondlib* or [foo], then creates an object that does something different, but calls it [foo]. Since *firstlib* is already being loaded first, then *firstlib*'s new [foo] will override *secondlib*'s well-established [foo], and everyone's patches break. [10]

2.4 When Pd-vanilla gains objectclasses

Every so often, Miller Puckette adds new objectclasses to Pd-vanilla, for example, [list] in 0.39, [sigmund~] in 0.40, [pd~], [stdout], and [pow~] in 0.42. This creates a situation very similar to was is outlined above with the fictional *firstlib* and *secondlib*. Recently, the addition of [pow~] to Pd-vanilla 0.42 in combination with the new objectclass overriding feature has illustrated the problems with relying on loading libraries into a global namespace. Pd-extended has always included *cyclone*'s [pow~] and loaded it into the global namespace by default. This [pow~] is a clone of the one included with Max/MSP, so its outlets are reversed as compared to the new [pow~] [8] in Pd-vanilla 0.42.

3. LEARNING FROM OTHERS

Pretty much all programming languages have a convention of putting the library/namespace loading functionality at the very beginning of the text file. Tcl programs usually start with `package require` statements and Python and Java programs start with `import` statements at the top of them. While there isn't a requirement for `package require` or `import` to be at the top, it is a strong convention to put those statements there. The logical equivalent in Pd would be the upper right corner of a patch, since execution in Pd flows from top to bottom, from right to left. Therefore it makes sense to place this library loading functionality in the upper right corner.

[2]Institute for Electronic Music, Graz, Austria

```
import_stmt     ::=  "import" module ["as" name] ( "," module ["as"
                     name] )*
                     | "from" relative_module "import" identifier
                     ["as" name]
                       ( "," identifier ["as" name] )*
                     | "from" relative_module "import" "("
                     identifier ["as" name]
                       ( "," identifier ["as" name] )* [","] ")"
                     | "from" module "import" "*"
module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
name            ::=  identifier
```

Figure 1: the complete grammar of Python's `import`

3.1 Python

Python is a programming language that has been designed with usability in mind. Python is also a language that includes well documented namespaces and library loading capabilities. Python also has a well used, documented, and discussed set of procedures for loading libraries based on the command `import`. Python's `import` provides great flexibility, but is also quite complicated. While the basic command is just `import X`, there is also `from X import *`, `from X import a,b,c`, and `X = __import__('X')`. [12] While this does provide lots of flexibility, it makes the `import` command itself quite elaborate in syntactic options, and makes the code itself perhaps more complicated since individual functions can be imported into the current namespace. It also separates the action of making the library available for use (i.e. `math.sin()` will run) and loading the library's functions into the local namespace (i.e. `sin()` is now mapped to `math.sin()`). In Python, the order of execution is top-to-bottom, then depth first. So if one file imports another which in turn imports the previous one, aka a "circular import", then the results can be unexpected. While it would be possible to attempt to solve this in the `import` code itself, Python relies on a convention against circular imports instead.[5]

3.2 Lua

In Lua, namespaces are created using the fundamental Lua data type: the table. This follows the philosophy of Lua, where the language should provide only a small set of "metamechanisms" which are then used to create things like namespaces and classes. In both these cases, Lua tables are used as the fundamental element. Additionally, these tables are treated as "first-class values" meaning that they can be handles like any other chunk of data, i.e. assigned to variables, passed to functions, etc.[3]

A Lua library is loaded into the global namespace using `require`. Once it is loaded, it can be used with its library prefix, e.g. `foo.myfunc()`. Many libraries in Lua are loaded by default, like the math or string libraries, so `require` is not needed before loading them. Whether a library is loaded by default or not is defined by what is compiled into Lua; it is also possible to compile a library into Lua so that it is loaded by default. Lua's packages and files that hold their code share the same name, which is enforced by convention

```
package require Tcl 8
package require ncgi 1
package provide javascript 1.0.2

namespace eval ::javascript {
    variable SelectionObjList {}
}
```

Figure 2: an example of Tcl's `package` and `namespace`

only. While this is not technically required, Lua provides the `_REQUIREDNAME` variable to make the package name automatically be derived from the file name.

The package name is then used as a prefix to that package's functions, so a `complex` package would have functions like `complex.add()`. If you want to use just the function name without the package name, then you need to declare a local variable with that name and then assign the function to it, i.e. `local add, i = complex.add, complex.i`. So instead of providing an explicit command for importing function names into the global namespace, Lua expects people to use the fundamental building blocks like local variables. [2]

To address problems caused by exporting package names into the global namespace, Lua's `import` was conceived. `import` built upon `require` and written in Lua itself, and is designed to use all local names for packages and their functions. Using `import` requires even the package names to be loaded into a local variable, so `complex.add` would not be available until `local complex = import "complex"` is run. [1] This behavior brings Lua closer to Python's `import` and namespaces.

3.3 Tcl

Tcl is not natively an object-oriented language, so the organization of code is quite different than Lua and Python. Libraries are sets of procedures (aka `procs`) which are gathered into packages organized around a similar idea. The concepts of "package" and "namespace" are separate in Tcl, with separate commands. The `package` and `namespace` commands are what Tcl uses for loading libraries and managing namespaces. [13] Other languages like Java and Python combine these two functions into a single `import` command. While this arrangement provides extraordinary flexibility, they are very complicated and are difficult for even a moderately skilled Tcl programmer to grasp. Usually the `package` and the `namespace` end up being the same entity, so this flexibility is not often used. This flexibility also seems to prevent the development of widely accepted standard idioms for using Tcl namespaces. There are so many ways to organize the code into libraries that most people end up using their own style.

One advantage of the Tcl approach to libraries is the relative simplicity of the arrangement on the file system. A set of procedures are put into a given namespace and included in a single file. These files can then be placed into a directory to be made into a package using `pkg_mkIndex`. The files included in a package can also be compiled binaries, so a single structure is used for creating packages whether the code is Tcl text or a compiled C binary.

Before Tcl namespaces became dominant, the Tcl community tried to solve the problem of name collisions using automated social techniques rather than features of the language itself. The NIST Identifier Collaboration Service (NICS) was organized as a central registry of names that was intended to be a central registrar of unique identifiers using in programming. The Tcl community also used unique prefixes for all commands in a library as a means for dealing with name conflicts, [4] a strategy that is used in some Pd libraries, and a wide array of Max/MSP libraries, including Jitter. Tcl namespaces have now replaced these techniques and the NICS has since been largely forgotten. Indeed the NICS website itself is no longer even running.[3]

4. A PROPOSED SOLUTION

With the explosion of code generated for Pd in the past decade, we now also face the same issues that triggered the introduction of namespaces to Python, Lua, and Tcl. There is much confusion and hassle created by name conflicts, and there isn't a clear technique to handle the conflicts. Also, there are many different formats for distributing and loading reusable collections of code, some of which are not compatible which each other.

4.1 Patch-local Namespaces

The core module of programs in Pd is the patch itself, also known as the canvas when talking about Pd's implementation. Pd patches are often used as reusable objects themselves. Therefore it makes sense to make the patch represent the most local level. Next, we need to define namespaces in terms of Pd: a set of symbols representing objectclasses which are available to be created as instances when typed into an object box in a patch canvas. [9] Currently, Pd has a global namespace and load path, and a load path that is local to the parent patch and affects any patch that is used in that parent patch. This load path effectively acts as a namespace for objects written in Pd since they are never cached when loaded. For binary objects, the load path only acts as a load path, and there is currently only the global namespace for loaded objectclasses.

Having each patch have its own namespace means that a given patch's library configuration can be embedded in the patch, thereby insuring that this patch will find the right libraries so matter how the parent patch or any other patch used in a project is setup. To be effective, the namespaces need to behave the same for all types of objectclasses, from binaries to abstractions. This means adding a patch-local namespace for loaded objectclasses that mirrors the patch-local load path. [7]

The current parent-local based load path is incomplete because there is no accompanying affect on the cache of loaded objectclasses. It could stay in place and be flushed out as another namespace level in between global and patch-local. This then adds a level of complexity without a clear benefit. Having just global and patch-local namespaces provides a lot of flexibility with a very simple and clean interface. The global settings are controlled via the application preferences and the patch-local settings are controlled by either [`import`] or [`declare`].

4.1.1 Python's `import` except simple

One of Pd's greatest strengths is its extremely simple syntax. Following that tradition it makes sense to have objects

[3]http://pitch.nist.gov/nics

related to the patch-local namespace also have an extremely simple syntax. Following that ideal, it is apparent that the only thing that people need to do with libraries into the patch-local namespace is load them. [`import firstlib secondlib`] already provides that functionality without any additional syntax. One additional feature that could be added to [`import`] is the ability to load individual objectclasses from a library by directly specifying it, e.g. [`import cyclone/pow~`] would load only [`pow~`] into the patch-local namespace, but not any other part of *cyclone*.

Tcl provides separate commands for loading libraries and registering names in the namespace, and Python provides detailed ways of loading names into the namespaces. This level of technical detail is unnecessary for all but the largest of large projects. Pd is not meant for very large software projects, while Python is willing to sacrifice some accessibility in order to allow for projects with very large codebases. In keeping with the simple nature of Pd, libraries should be automatically loaded on demand, and only namespaces should be explicitly manipulated. This means that there needs to be a separation of the namespace code and the library loading code. The only thing the user needs to know is whether the library is listed in an [`import`] statement in the current patch. If so, the library is loaded, if not, the library is not loaded.

4.1.2 *a* `distro` *method*

There are many different distros of Pd, and most of them have a custom set of objectclasses that are available in the global namespace. Pd-vanilla has the standard set, with a few more in the "extra" folder. Pd-extended has a many libraries loaded by default in the global namespace. RjDj is a new distro that also has a custom set of objectclasses in the global namespace. It is a natural impulse to have a preconfigured global namespace with Pd since Pd and Max have generally always had only a single global namespace. This can cause problems when a patch is created on one distro yet a different set of objects are available on another distro. I propose to add a `-distro` method to the `#X declare` functionality in the patch. Each distro of Pd would save this information into every patch. While this does add a bit of complexity to handling libraries, it would make starting out with Pd a much easier experience.

When Pd encountered this distro tag, it would set up the environment to limit the available libraries to the Pd-vanilla objects. "pd-extended" would set up the default libraries that are loaded in the current Pd-extended. For example, if you created [`pow~`] with the "pd-extended" distro, you'd get [`cyclone/pow~`]. Versions could also be included, so with something like "vanilla-0.41.4", then [`pow~`] would fail to create, but with "vanilla" you'd get the most recent version, including the new [`pow~`] introduced in 0.42. [6]

4.2 A Common Library Format

In order for the namespaces to be fully useful, it is necessary to organize all of the code into distinct libraries, since the library is the core unit of organizing namespaces. The library format should be simple and common across all implementation methods. So objectclasses implemented in C, Pd, Lua, etc should all be able to be included in a single library. This also means that the core of Pd should include as few objectclasses as possible, and the ones that are part of Pd-vanilla should be separated out into its own library. Help and example patches should also be included in the same library. Lastly, it should be easy to install and load.

This is mostly possible with the current implementation of libdirs: C and Pd objectclasses can co-exist, the help patches can be included as well. For including example patches, there is a PDDP proposal to use a standard postfix "-example.pd" mirroring the standard "-help.pd" for help patches. One key part that is missing is the ability to support shared code among objectclasses within a library.

4.2.1 *Code shared within a library*

Since Pd and its implementation lacks a object-oriented class hierarchy, it is often useful to have an internal library of code that is used throughout the implementation of a Pd library. *Gem* and *PDP* are examples of this, where the objectclasses are broken out into separate .c files, one-per-class, but then they share a standard chunk of code between them all. This practice has also proven useful Lua and Tcl implementations of Pd libraries. A good library format should then support this practice.

In this proposal, this shared implementation code is handled by a shared library included in the library itself. It has a standardized name so that the loader knows which file to load. When the library is loaded, this shared implementation library is loaded first before any of the individual object classes. Then when each objectclass is loaded from its individual file, the shared code is already available. The standard name proposed here is the filename prefix `lib` along with the appropriate file extension for that file on the given operating system. For binaries, that would be `.so` for GNU/Linux, `.dylib` for Mac OS X, and `.dll` for Windows. For example, *Gem* would then include a `libgem.so` on GNU/Linux, and a library written in Lua called *foo* would include a `libfoo.lua` on all platforms.

Also, to represent Pd's split between the `pd` and `pd-gui` processes, there should be a separate library prefix for GUI related code. For example, *tkwidgets* would include `libguitkwidgets.`

4.3 Search order is also important

In an effort to make the library format as simple as possible, it is also necessary to consider the order that the library paths are searched for a given name. Currently, Pd searches each possible name through all of the paths before it tries the next type. This means it is not possible to use the name that binary object has if that binary is included anywhere in the search path. The binary always takes precedence, even if someone sticks a Pd patch with that name in the same folder as the project (e.g. `"."`) While is it a good idea to avoid name clashes, it is confusing if you create a patch with a name, and then something else is loaded from the path. With more and more Pd libraries being written in Pd (e.g. abstractions), Lua, and other languages, we should consider treating all of these implementation methods as equals.

5. GETTING IT DONE

The ideas presented in this paper have been debated for many years now, and some of them have been implemented and tested. The next step is to more fully implement namespaces and a common library format and test them in real world conditions. There are three parts to this puzzle: a common library format, namespace support, and search order. They can be implemented and tested individually, and I plan on starting with the common library format and

namespace support. The ideas for the search path and `#X declare -distro` are more raw, so they should be discussed and tested more before committing to them.

6. ACKNOWLEDGMENTS

It is important to note that while I am the sole author of this paper, I do not make a claim to the ideas outlined here. This solution is built upon the ideas and work of many people such as IOhannes m. zmoelnig, Frank Barknecht, Miller Puckette, Roman Haefeli, pdmtl, Luke Iannini, Marius Schebella, and many long discussions on the mailing lists, IRC, and in person. This paper is the end result of my cataloging of the results of these discussions. I am sure I have left out other contributors, please accept my apology in advance.

7. REFERENCES

[1] W. Couwenberg. Technical note 11: Require revisited: Import. Technical report, Lua.org, February 2003.

[2] R. Ierusalimschy. Technical note 7: Modules and packages. Technical report, Lua.org, August 2002.

[3] R. Ierusalimschy. Programming in lua. `http://www.lua.org/pil/`, 2004.

[4] D. Libes. Managing Tcl's namespaces collaboratively. In *Proceedings of the 5th conference on Annual Tcl/Tk Workshop*, volume 5. USENIX Association, 1997.

[5] F. Lundh. Importing python modules. `http://effbot.org/zone/import-confusion.htm`, 2001.

[6] pd-dev. [declare -distro vanilla]. `http://lists.puredata.info/pipermail/pd-dev/2009-03/013165.html`, March 2009.

[7] pd-list. Abstractions search path hirarchy. `http://lists.puredata.info/pipermail/pd-list/2008-06/063294.html`, June 2008.

[8] pd-list. Cyclone in vanilla? `http://lists.puredata.info/pipermail/pd-list/2008-04/061603.html`, April 2008.

[9] pd-list. declare [looooooooooong]. `http://lists.puredata.info/pipermail/pd-list/2008-07/064267.html`, July 2008.

[10] pd-list. pd-ext paths, libs and help. `http://www.mail-archive.com/pd-list@iem.at/msg16238.html`, March 2008.

[11] H.-C. Steiner. libdir - a new format for pd libraries. `http://puredata.info/docs/developer/Libdir`, 2007.

[12] G. van Rossum. Python reference manual. `http://www.python.org/doc/2.5.2/ref/`, February 2008.

[13] B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1997.