

# Getting Programs and Libraries to Work with Piper

Brad Chapman (chapmanb@arches.uga.edu)

Last Update–5 June 00

## Contents

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>What does it mean to “plug” a program into Piper?</b>                 | <b>1</b> |
| <b>2</b> | <b>The Overall Picture</b>   | <b>1</b> |
| 2.1      | The different Parts of Piper . . . . .                                   | 1        |
| 2.2      | Two different ways to get a program working. . . . .                     | 2        |
| <b>3</b> | <b>An example program</b>  | <b>2</b> |
| 3.1      | How the example programs works normally . . . . .                        | 2        |
| <b>4</b> | <b>The Hard Way–Getting everything working by hand</b>                   | <b>2</b> |
| 4.1      | Using existing “loci” already working in Piper. . . . .                  | 2        |
| 4.2      | Making the program a unique “base” locus. . . . .                        | 3        |
| 4.2.1    | Writing the .cls file describing the program . . . . .                   | 3        |
| 4.2.2    | Putting the .cls file in the right place so Piper will find it . . . . . | 4        |
| 4.2.3    | Registering a program or library with the brokering layer . . . . .      | 4        |
| 4.2.4    | Making a program or library loadable in the processing layer . . . . .   | 4        |
| <b>5</b> | <b>The Easy Way–using Piper itself as a development environment</b>      | <b>4</b> |

## 1 What does it mean to “plug” a program into Piper?

A major goal of Piper is to provide a standard way to connect and manipulate programs and libraries. In order to meet this goal, a particular program has to be able to be manipulated using Piper tools. This requires that Piper is able to recognize the program and know how to deal with it. Thus, getting a program or library “plugged” into Piper requires that you tell Piper what is going on with the program, and tell it how to load and run it. This should not require any changes to the program itself, so that Piper will be able to “plug in” and work with a wide variety of programs.

## 2 The Overall Picture

As I just mentioned, the basic requirement for getting a program working with Piper is to give Piper enough information to know how to deal with it. Well, duh. That isn’t too hard to figure out. The hard part is, how do I do this? . . . Well, each part of Piper will have specific requirements for dealing with the program or library, and so each part will need to be informed about what is going on with it. Let’s take a look at the parts step by step.

### 2.1 The different Parts of Piper

1. The User Interface Layer (UIL) - The UIL is designed to be a “dumb” front end to the whole Piper system, so you don’t need to tell it anything in order to get your program working with it. Well, that one was easy :-)

2. The Definition Layer (DL) - The DL speaks with the UIL and helps it connect your programs to other programs in the correct way. So, the DL will need to know information about what kind of inputs and outputs your program has, and what other parameters it needs to work. This information will be supplied in a `.cls` file with your program's name in front of it. This `.cls` file is in XML format, and more on this follows.
3. The Brokering Layer (BL) - The BL is responsible for implementing a “work-flow diagram” (ie. a group of programs and libraries which are connected together in order to perform a particular task) in an efficient manner. Thus it will need information about XXX?? (Jarl?)
4. The Processing Layer (PL) - The PL is responsible for actually running programs and libraries, and also in dealing with shuttling information between the programs. So it will need information about how to load and run the program, and... (XXXmore?–Jean-Marc?)

## 2.2 Two different ways to get a program working.

Basically, if you are thinking about getting something working with Piper, there are two different cases you will run into:

1. The program can be built using already available Piper loci - This is by far the easiest case, and should hopefully be possible for a number of command line programs. In this situation, you will use already available Piper loci to build up your program.
2. The program needs to be a “base” Piper locus - This will be the case if your program has such strange input and outputs or unusual settings, that no other combination of existing loci can model it properly.

## 3 An example program

For going through the process of plugging in a program, we are going to use the simple example of the UNIX utility `kill`, which is, as the name suggests, used to destroy (or signal) processes which are out of control or locked up. I have a strange fondness for this utility on account of the many times I've used it during hacking on Piper, and hopefully it is widely known enough to make sense.

### 3.1 How the example programs works normally

The use of `kill` is relatively straightforward, which is another reason why I picked it (less to explain! yay!). Let's say that things have gone horribly wrong with Piper (of course, this is completely hypothetical—that would never happen in real life :-), and you need to shut it down. Also assume that it is running under the process id 1234. So, the most basic way to kill it would be:

```
$ kill 1234
```

So `kill` accepts an argument, which is the process id to be killed. It also accepts a couple of options, such as the signal name to send to the process. Let's say that Piper has died so horribly, that a regular `TERM` signal won't kill it, then we need to send:

```
$ kill -9 1234
```

to make sure it is good and dead. So you've got the overall picture. `kill` has a argument (the process id to kill) and some options, one of which is the signal number option, shown above. But `kill` also has two outputs, namely text which is output to the screen, and the return value of `kill`, which indicates whether or not it is successful. Now that we understand `kill`, lets move on to getting it working with Piper.

## 4 The Hard Way—Getting everything working by hand

### 4.1 Using existing “loci” already working in Piper.

Need to figure out how this would be done “by hand.” `kill` could be done using `NetExec`, but we need to figure out the XML format for this.

## 4.2 Making the program a unique “base” locus.

Okay, so let’s pretend that `kill` is so complex that we can’t model it using the basic Piper loci. Let’s go ahead and make it a “base” locus.

### 4.2.1 Writing the `.cls` file describing the program

The first step is to create a unique XML file describing the `kill` locus we are creating. By default, these files are labelled with a `.cls` extension in Piper, and each file must have a unique name. So we’ll call ours `kill.cls`. The strict format of these `.cls` files is described in the Document Type Definition (DTD) file `locusPlugin.dtd`. Basically, what this file says is that a locus has the following components:

1. `parameter` - This is an essential component of the locus.
2. `input` - This is an input into the locus.
3. `output` - This is an output from the locus.

For our `kill` example, the PID to kill is a parameter, the inputs are the options that can be passed, and the outputs are the text output to the screen and the status of a process. So let’s look at a completed `kill.cls` file:

```
<!DOCTYPE locus SYSTEM "../dtd/locusPlugin.dtd">

<locus name = "kill" category = "UNIX">

  <parameter name = "PID" type = "P_STRING" expand = "yes">

  <input name = "argument" id = "0" type = "P_STRING"/>

  <output name = "command_output" id = "0" type = "P_STRING" />
  <output name = "command_status" id = "1" type = "P_STRING" />

  <comment>
  Terminate or signal a process.
  </comment>

</locus>
```

The first element is the `<!DOCTYPE>` tag, which specifies the type of XML file we are creating. In our case, we are creating a `locus` type, whose definition is supplied in the file `locusPlugin.dtd`.

The next element is the main definition of our locus. Here we specify the name (`kill`) and the category under which it will be grouped. For this case, we’ll just group this in the `UNIX` category. Note that categories are only for convenience in holding different types, each plugin to Piper must have a unique name.

Now we are ready to get into actually defining some stuff about the program. The first thing we encounter is a parameter, the “PID” that will be passed to `kill`. The `name` attribute specifies that this is the function of this parameter. This name should be unique for all of the different parameters, and should also be descriptive so that it can be used by a user interface as a clue to what the user should be entering for this value. The `type` attribute specifies the type of attribute that should be found here (XXX we need to decide on all the possible types we can have here). In this case, we expect a string. Finally, the attribute `expand` indicates whether or not this parameter can be duplicated. For instance, in the case of `kill`, you can specify multiple PIDs on the command line. So we could have a number of “PID” parameters, in a particular `kill` command.

Whew, that was so much fun that we are now ready to move on to the inputs and outputs. These are identical, so we can deal with them together. The `name` attribute is the same as we saw with parameters, and just specifies a simple unique string that identifies the type of input or output. The `type` is also the same as before, and specifies the type of connection that will be created (XXX Need to decide on all the types here as well). This is used so that connections

can only be created between “compatible” types. The `id` attribute is new, and it is used to uniquely specify each input or output, so that we can distinguish them all. This should be a number which is unique amongst all the connectors of a particular type (input or output). Inputs and outputs can also have an `expand` attribute, as we saw with the parameter example above.

The final item is the comment. This is an area used to specify information about the locus which could be displayed as a help command in the user interface.

Well, that’s it. Now you know everything there is to know about writing a `.cls` file!

#### **4.2.2 Putting the `.cls` file in the right place so Piper will find it**

Hmmm... Need to think about how this will work. Should have a default location in the `HOME` directory of the user where these can be put.

#### **4.2.3 Registering a program or library with the brokering layer**

Not positive how this’ll work. Jarl?

#### **4.2.4 Making a program or library loadable in the processing layer**

Not sure exactly how this will work, especially for libraries not written in C/C++. I’ll have to have Jean-Marc explain it to me...

## **5 The Easy Way—using Piper itself as a development environment**

Sorry, this isn’t functional yet, you’ll have to do it the hard way :-j