

How To Write a User Interface for Piper

Brad Chapman (chapmanb@arches.uga.edu)

Last Update–5 June 00

Contents

1	Where does the User Interface fit in?	1
2	The Structure of User Interface communication with the Definition Layer	3
3	The first step–registering with the definition layer	3
4	Adding things and messing around with them	4
4.1	The Document Interface	4
4.2	The Network Interface	5
5	Manipulating and extracting info from Loci	5
5.1	The Connector Interface	6
5.2	The Parameter Interface	7
5.3	Back to the Locus	7
6	Dealing with processes	7

1 Where does the User Interface fit in?

The User Interface is the most visible component of Piper, and is therefore the most important to the standard user. However, when implementing a user interface, it is important to understand how the user interface layer (UIL) relates to the other, less visible parts of the system. Figure 1 illustrates the relationships between the different Piper components.

The four major components of Piper are:

1. The User Interface Layer (UIL) - This is what we are talking about writing here, so hopefully you know what this is :-).
2. The Definition Layer (DL) - This is the only layer that the UIL communicates with. It stores all of the information about loci available to be manipulated in the user interface, provides permanent storage for work-flow diagrams created in the user interface, and generally allows the uil to communicate with Piper. In sum, the user interface should be able to do anything it desires by communicating with the definition layer.
3. The Brokering Layer (BL) - This layer provides functionality for executing connections between programs in an efficient manner and generally for maintaining an orderly flow through a created work flow diagram.
4. The Processing Layer (PL) - This layer loads and executes all programs and libraries that work within Piper.

So, the place of the User interface in all of this is to provide the essential interface by which a user can control the system. All of this is done by communication with the definition layer through the interfaces defined in the IDL (interface definition language) file `uil2dl.idl`.

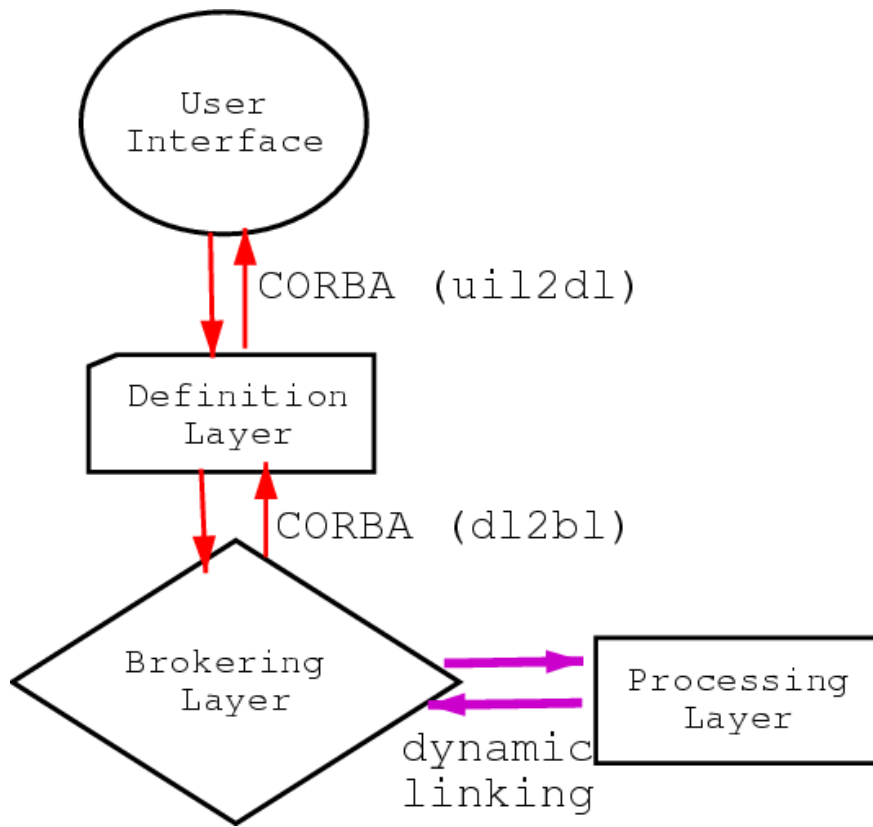


Figure 1: Relationships between different Piper components.

2 The Structure of User Interface communication with the Definition Layer

As we just hinted at in the last section by mentioning IDL files, the communication between the user interface and definition layer occurs through a CORBA (Common Object Request Broker Architecture) interface. Because of this, the user interface can be implemented in any programming language which has CORBA bindings (and a whole lot do). <http://www.corba.org> is a good place to get started if you want to learn more about how CORBA works.

Although the following is written so that it will hopefully be usable in any language, it is definitely python centric since that's the language I understand CORBA in the best. Implementations in other languages are **very** welcome, and I, for one, would be very happy to play with just about any language someone wanted to write a User interface in.

As mentioned previously, the interfaces defining communication between the user interface layer and the definition are located in the file `uil2dl.idl`. The aim of the rest of this is to make what is happening in that file comprehensible.

The file defines a number of objects which we'll be messing around with. These are:

1. `DIServer` - This is the entry point to connecting with the definition layer.
2. `UilClient` - This is an interface that must be implemented by all user interfaces, and allows the definition layer to verify a connection by a UI, and also to check that it is still alive.
3. `Document` - This is the largest interface describing an actual object that you work with. The document is the storage item that holds everything you do within Piper.
4. `Locus` (plural = `Loci`) - A locus is the central element in Piper. It represents a program or library or generally anything that does something. The name is also equivalent to "node," but locus is used for historical reasons, and also to prevent serious namespace clashes with DOM.
5. `Network` - A network is the platform on which you manipulate programs and libraries (loci). It is used for creating and deleting loci, as well as connecting them. So while a locus "does something" a network allows you to connect loci together to "do something useful." Note that a Network is also a locus, which allows networks to be nested within each other (sub-networks).
6. `Connector` - An object which connects two loci together. These are elements that "belong" to a locus.
7. `Parameter` - A information item that is part of a locus. This allows specific information to be set relating to a locus.
8. `Process` - This represents a series of connected loci (a work-flow diagram) that has been submitted for processing, and allows information about a running process to be queried.

Hopefully it makes sense how these objects described above relate to the definitions in the IDL file. These objects are everything that we'll be dealing with to manipulate the internal workings of Piper.

3 The first step—registering with the definition layer

Before we can start messing around with stuff like manipulating inner workings, we need to be able to connect and authenticate the user interface with the definition layer. This occurs in a two step process. First, the user interface must supply a server which implements the following simple interface:

```
interface UilClient {
    readonly attribute string username;
    readonly attribute string password;

    void ping();
};
```

So the client needs to provide implementations for three basic functions. The first two are the username and password that are being used to connect with the definition layer. These are used for authentication and to provide a simple kind of security system.

The second thing to define is `ping()`, which is a do nothing function, so it is really easy to implement :-). The purpose of `ping()` is that the definition layer will call it periodically to check that the user interface has not crashed. This provides a way to keep the definition layer server in tune with what's going on in the user interfaces.

After implementing this interface, it needs to be submitted to the definition layer server (so the server can call it's methods). In order to make this submission, the user interface first needs to get a hold of an object that has the submission interface. The way this is done in Piper is that the definition layer, upon starting up, writes its IOR (Initial Object Representation—XXXis this right? I just made up the acronym meaning:;) to the file `$HOME/piper_info/ior/uil2dl.ior` in a stringified format. So the way to connect to the definition layer is to read the contents of this file into a string, and then perform a call like `orb.string_to_object(the_ior_string)`, which will convert the string IOR into an object representing the definition layer server.

The object that you'll get implements the following interface with a single function:

```
interface DlServer {
Document addDocument(in UilClient clientInfo);
};
```

The function `addDocument()` allows you to submit the user interface client described above, and get back a document object, from which you can start actually doing something!

4 Adding things and messing around with them

4.1 The Document Interface

The document object that your receive has the following interface:

```
interface Document {

PluginList getPlugins();
void addPlugin(in Plugin pluginToAdd);

readonly attribute LocusList selectedLoci;

void clearSelectedLoci();
void setSelectedLoci(in Locus locusToSet);
void addSelectedLoci(in Locus locusToAdd);

Network addNetwork();

Process run();

void close();
};
```

The functions `clearSelectedLoci()`, `setSelectedLoci()`, `addSelectedLoci()`, and `run()`—, all deal with setting up a list of loci to be submitted as a process. We can ignore these for now, since they will be discussed layer.

The first thing you probably want to use is the `getPlugins()` function, which will return a list of all of the program and library plugins which are available to be used in piper. The plugins are returned as the following struct:

```
struct Plugin {
string category;
```

```
string name;
};
```

`name` is the unique name of the plugin, and `category` is a period separated list of names describing the category the plugin falls under (ie. the unix program `'ls'` might be categorized under `unix.utilities`). These names are all of the different types of loci which are available to be created.

The second important thing to call is `addNetwork()`, which actually adds a network that you can do things with (XXXToDo - I need to talk with Jean-Marc and see about adding a type attribute so we can support iterators, etc.). This network will allow you to start doing the interesting work of dealing with loci.

4.2 The Network Interface

The network object that you get back from a `addNetwork()` call to the document is the following:

```
interface Network : Locus {
  Locus addLocus(in string type) raises (TypeError);

  Network addNetwork();

  void removeLocus(in Locus locusToDel) raises (LookupError);

  void connectLoci(in Locus inLocus, in Connector inConnector,
    in Locus outLocus, in Connector outConnector)
    raises (TypeError);
  void disconnectLoci(in Locus inLocus, in Connector inConnector,
    in Locus outLocus, in Connector outConnector);

};
```

One very important thing to notice immediately is that the `Network` object inherits from the `Locus` objects, so it also has all of the functions of a `Locus`. This is important to remember, but for right now we'll just deal with the `Network` only functions of the `Network` object.

The functions the `Network` implements are pretty straightforward, since they are things you are actually going to be doing the most of: creating loci and connecting them together. The `addLocus()` interface takes a string argument type. This type should be one of the names that you got when you used the document interface to get all of the plugins. So this allows you to add any available type of locus to the current workspace. Loci can, of course, also be removed by passing them to the function `removeLocus`.

The `Network` interface also allows you to connect and disconnect particular `Connectors` of a `Locus` through the `connectLocus()` and `disconnectLocus()` functions, respectively.

So, we now know enough to be able to add loci and connect them together. However, our goal is to be able to do this in a way that makes some sense, so that we can accomplish a useful task. To achieve this extremely lofty goal, we'll now need to begin to look at how to get information from a locus.

5 Manipulating and extracting info from Loci

When a `Network` adds a locus of a particular type, it will get back an object with the following interface:

```
interface Locus {

  readonly attribute ConnectorList inputs;
  readonly attribute ConnectorList outputs;
  readonly attribute ParameterList parameters;

  void addInput(in Connector input);
```

```

void addOutput(in Connector output);
void addParameter(in Parameter param);

void removeInput(in Connector input);
void removeOutput(in Connector output);
void removeParameter(in Parameter param);

void setPosition(in long x, in long y);

void save(in string saveModule, in string saveName);

};

```

This is a big interface (XXX and definitely not everything works yet), but right now we are going to focus on the three most useful things, the attributes `inputs`, `outputs` and `parameters`. Each of these defines a list of inputs, output, and parameters (obviously :-)) that are associated with the locus. Each of these objects provides information about itself, and it is this information that will allow you to make useful connections.

5.1 The Connector Interface

For both the inputs and outputs of a locus, we will be dealing with the connector interface.

```

interface Connector {
enum ConnectorType {
P_VALUE, P_STREAM, P_OBJECT, P_CONDITION, P_STRING, P_ANY
};

attribute string name;
attribute long id;
attribute ConnectorType type;
};

```

Each connector has three attributes. The `name` is a string describing the connector, the `id` is a unique number for the connector, and the `type` is the type of connector. Currently, a connector can be any of the following types (note, the `P_` is just a “namespace” thing to prevent stuff like `STRING` and `ANY` from clashing with defined corba types):

1. `VALUE` - A connector that passes a number value.
2. `STREAM` - Passes a (file) stream.
3. `OBJECT` - Passes an object (so I guess this can only work between loci that are doing things in the same language).
4. `CONDITION` - Passes a particular condition for doing things (I don't understand how this works...)
5. `STRING` - Passes a string.
6. `ANY` - A catch all category to describe connectors that can accept lots of different types of objects.

So by accessing these three different attributes of each connector, you can define how things should be connected, and also provide some information to the user to help them connect things properly.

5.2 The Parameter Interface

The parameter interface defines different information items which are an inherent part of a locus. This is kind of broad on purpose, so that lots of different parameters can be represented through a single interface.

```
interface Parameter {
  enum ParameterType {
    P_VALUE, P_STRING, P_ANY
  };

  attribute string name;
  attribute ParameterType type;

  void setInfo(in any info) raises (TypeError);

  any getInfo() raises (LookupError);
};
```

Like a connector, a parameter has name and type attributes. For a parameter, the following attribute types are defined:

1. VALUE - A numerical value.
2. STRING - An information string.
3. ANY - A huge catch-all category for all of the tons of different parameter types that can be held.

These attributes provide some information about the parameter, so that a user can be able to set it properly.

However, the major purpose of a parameter is to store and set information related to a locus, and this is deal with using the functions `setInfo()` and `getInfo()`. These functions use with `CORBA::any`, so they provide a mechanism to store and deal with tons of different types of data. Hopefully, this polymorphism will allow many different locus types to be easily usable under this framework.

5.3 Back to the Locus

Now that the major attributes of a locus make sense, we are faced with the most important task of all—displaying the information on a locus in a way that will allow the user to make good decisions when connecting loci, and thus create useful and powerful diagrams of connected loci. This of course, is completely up to the creativity of the user interface designer. This framework is meant to be relatively simple and straightforward so that it doesn't constrain the development of the user interface to a particular format. However, it is also meant to be structured and powerful enough to make it easy to deal with the definition layer and extract all of the information it has to offer.

6 Dealing with processes

We're not quite ready for this yet.