Using ILU with Python: A Tutorial

Bill Janssen <janssen@parc.xerox.com>

Formatted 7 May 1996.

0.1 Introduction

This tutorial will show how to use the ILU system with the programming language Python, both as a way of developing software libraries, and as a way of building distributed systems. In an extended example, we'll build an ILU module that implements a simple four-function calculator, capable of addition, subtraction, multiplication, and division. It will signal an error if the user attempts to divide by zero. The example demonstrates how to specify the interface for the module; how to implement the module in Python; how to use that implementation as a simple library; how to provide the module as a remote service; how to write a client of that remote service; and how to use subtyping to extend an object type and provide different versions of a module. We'll also demonstrate how to use OMG IDL with ILU, and discuss the notion of network garbage collection.

Each of the programs and files referenced in this tutorial is available as a complete program in a separate appendix to this document; parts of programs are quoted in the text of the tutorial.

0.2 Specifying the Interface

Our first task is to specify more exactly what it is we're trying to provide. A typical four-function calculator lets a user enter a value, then press an operation key, either +, -, /, or *, then enter another number, then press = to actually have the operation happen. There's usually a CLEAR button to press to reset the state of the calculator. We want to provide something like that.

We'll recast this a bit more formally as the *interface* of our module; that is, the way the module will appear to clients of its functionality. The interface typically describes a number of function calls which can be made into the module, listing their arguments and return types, and describing their effects. ILU uses *object-oriented* interfaces, in which the functions in the interface are grouped into sets, each of which applies to an *object type*. These functions are called *methods*.

For example, we can think of the calculator as an object type, with several methods: Add, Subtract, Multiply, Divide, Clear, etc. ILU provides a standard notation to write this down with, called *ISL* (which stands for "Interface Specification Language"). ISL is a declarative language which can be processed by computer programs. It allows you to define object types (with methods), other non-object types, exceptions, and constants.

The interface for our calculator would be written in ISL as:

```
INTERFACE Tutorial;

EXCEPTION DivideByZero;

TYPE Calculator = OBJECT
  METHODS
    SetValue (v : REAL),
    GetValue () : REAL,
    Add (v : REAL),
    Subtract (v : REAL),
    Multiply (v : REAL),
    Divide (v : REAL) RAISES DivideByZero END
  END;
```

This defines an interface `Tutorial`, an exception `DivideByZero`, and an object type `Calculator`. Let's consider these one by one.

The interface, `Tutorial`, is a way of grouping a number of type and exception definitions. This is important to prevent collisions between names defined by one group and names defined by another group. For example, suppose two different people had defined two different object types, with different methods, but both called `Calculator`! It would be impossible to tell which calculator was meant. By defining the `Calculator` object type within the scope of the `Tutorial` interface, this confusion can be avoided.

The exception, `DivideByZero`, is a formal name for a particular kind of error, division by zero. Exceptions in ILU can specify an *exception-value type*, as well, which means that real errors of that kind have a value of the exception-value type associated with them. This allows the error to contain useful information about why it might have come about. However, `DivideByZero` is a simple exception, and has no exception-value type defined. We should note that the full name of this exception is `Tutorial.DivideByZero`, but for this tutorial we'll simply call our exceptions and types by their short name.

The object type, `Calculator` (again, really `Tutorial.Calculator`), is a set of six methods. Two of those methods, `SetValue` and `GetValue`, allow us to enter a number into the calculator object, and "read" the number. Note that `SetValue` takes a single argument, *v*, of type `REAL`. `REAL` is a built-in ISL type, denoting a 64-bit floating point number. Built-in ISL types are things like `INTEGER` (32-bit signed integer), `BYTE` (8-bit unsigned byte), and `CHARACTER` (16-bit Unicode character). Other more complicated types are built up from these simple types using ISL *type constructors*, such as `SEQUENCE OF`, `RECORD`, or `ARRAY OF`.

Note also that `SetValue` does not return a value, and neither do `Add`, `Subtract`, `Multiply`, or `Divide`. Rather, when you want to see what the current value of the calculator is, you must call `GetValue`, a method which has no arguments, but which returns a `REAL` value, which is the value of the calculator object. This is an arbitrary decision on our part; we could have written the interface differently, say as

```
TYPE NotOurCalculator = OBJECT
  METHODS
    SetValue () : REAL,
    Add (v : REAL) : REAL,
    Subtract (v : REAL) : REAL,
    Multiply (v : REAL) : REAL,
    Divide (v : REAL) : REAL RAISES DivideByZero END
  END;
```

– but we didn't.

Our list of methods on `Calculator` is bracketed by the two keywords `METHODS` and `END`, and the elements are separated from each other by commas. This is pretty standard in ISL: elements of a list are separated by commas; the keyword `END` is used when an explicit list-end marker is needed (but not when it's not necessary, as in the list of arguments to a method); the list often begins with some keyword, like `METHODS`. The *raises clause* (the list of exceptions which a method might raise) of the method `Divide` provides another example of a list, this time with only one member, introduced by the keyword `RAISES`.

Another standard feature of ISL is separating a name, like `v`, from a type, like `REAL`, with a colon character. For example, constants are defined with syntax like

```
CONSTANT Zero : INTEGER = 0;
```

Definitions, of interface, types, constants, and exceptions, are terminated with a semicolon.

We should expand our interface a bit by adding more documentation on what our methods actually do. We can do this with the *docstring* feature of ISL, which allows the user to add arbitrary text to object type definitions and method definitions. Using this, we can write

```
INTERFACE Tutorial;

EXCEPTION DivideByZero
  "this error is signalled if the client of the Calculator calls
the Divide method with a value of 0";

TYPE Calculator = OBJECT
  COLLECTIBLE
  DOCUMENTATION "4-function calculator"
  METHODS
    SetValue (v : REAL) "Set the value of the calculator to `v'",
    GetValue () : REAL  "Return the value of the calculator",
    Add (v : REAL)      "Adds `v' to the calculator's value",
    Subtract (v : REAL) "Subtracts `v' from the calculator's value",
    Multiply (v : REAL) "Multiplies the calculator's value by `v'",
    Divide (v : REAL) RAISES DivideByZero END
      "Divides the calculator's value by `v'"
  END;
```

Note that we can use the `DOCUMENTATION` keyword on object types to add documentation about the object type, and can simply add documentation strings to the end of exception and method definitions. These docstrings are passed on to the Python docstring system, so that they are available at runtime from Python. Documentation strings cannot currently be used for non-object types.

ILU provides a program, `islscan`, which can be used to check the syntax of an ISL specification. `islscan` parses the specification and summarizes it to standard output:

```
% islscan Tutorial.isl
Interface "Tutorial", imports "ilu"
  {defined on line 1
   of file /tmp/tutorial/Tutorial.isl (Fri Jan 27 09:41:12 1995)}

Types:
  real                        {<built-in>, referenced on 10 11 12 13 14 15}

Classes:
  Calculator                  {defined on line 17}
    methods:
      SetValue (v : real);                          {defined 10, id 1}
        "Set the value of the calculator to `v'"
      GetValue () : real;                           {defined 11, id 2}
        "Return the value of the calculator"
      Add (v : real);                               {defined 12, id 3}
        "Adds `v' to the calculator's value"
      Subtract (v : real);                          {defined 13, id 4}
        "Subtracts `v' from the calculator's value"
      Multiply (v : real);                          {defined 14, id 5}
        "Multiplies the calculator's value by `v'"
      Divide (v : real) {DivideByZero};             {defined 16, id 6}
        "Divides the calculator's value by `v'"
    documentation:
      "4-function calculator"
    unique id:  ilu:cigqcW09P1FF98gYVOhf5XxGf15

Exceptions:
  DivideByZero                {defined on line 5, refs 15}
%
```

`islscan` simply lists the types defined in the interface, separating out object types (which it calls "classes"), the exceptions, and the constants. Note that for the `Calculator` object type, it also lists something called its *unique id*. This is a 160-bit number (expressed in base 64) that ILU assigns automatically to every type, as a way of distinguishing them. While it might interesting to know that it exists (:-), the ILU user never has know what it is; `islscan` supplies it for the convenience of the ILU implementors, who sometimes do have to know it.

0.3 Implementing the True Module

After we've defined an interface, we then need to supply an implementation of our module. Implementations can be done in any language supported by ILU. Which language you choose often depends on what sort of operations have to be performed in implementing the specific functions of the module. Different languages have specific advantages and disadvantages in different areas. Another consideration is whether you wish to use the implementation mainly as a library, in which case it should probably be done in the same language as the rest of your applications, or mainly as a remote service, in which case the specific implementation language is less important.

We'll demonstrate an implementation of the `Calculator` object type in Python, which is one of the most capable of all the ILU-supported languages. This is just a matter of defining a Python class, corresponding to the `Tutorial.Calculator` type. Before we do that, though, we'll explain how the names and signatures of the Python functions are arrived at.

0.3.1 What the Interface Looks Like in Python

For every programming language supported by ILU, there is a standard *mapping* defined from ISL to that programming language. This mapping defines what ISL type names, exception names, method names, and so on look like in that programming language.

The mapping for Python is straightforward. For type names, such as `Tutorial.Calculator`, the Python name of the ISL type `Interface.Name` is `Interface.Name`, with any hyphens replaced by underscores. That is, the name of the interface in ISL becomes the name of the module in Python. So the name of our `Calculator` type in Python would be `Tutorial.Calculator`, which is really the name of a Python class.

The Python mapping for a method name such as `SetValue` is the method name, with any hyphens replaced by underscores. The return type of this Python method is whatever is specified in the ISL specification for the method, or `None` if no type is specified. The arguments for the Python method are the same as specified in the ISL; their types are the Python types corresponding to the ISL types, *except* that one extra argument is added to the beginning of each Python version of an ISL method; it is an *instance* of the object type on which the method is defined. An instance is simply a value of that type. Thus the Python method corresponding to our ISL `SetValue` would have the prototype signature

```
def SetValue (self, v):
```

Similarly, the signatures for the other methods, in Python, are

```
def GetValue (self):

def Add (self, v):

def Subtract (self, v):

def Multiply (self, v):

def Divide (self, v):
```

Note that even though the `Divide` method can raise an exception, the signature looks like those of the other methods. This is because the normal Python exception signalling mechanism is used to signal exceptions back to the caller. The mapping of exception names is similar to the mapping used for types. So the exception `Tutorial.DivideByZero` would also have the name `Tutorial.DivideByZero`, in Python.

One way to see what all the Python names for an interface look like is to run the program `python-stubber`. This program reads an ISL file, and generates the necessary Python code to support that interface in Python. One of the files generated is '*Interface*`.py`', which contains the definitions of all the Python types for that interface.

```
% python-stubber Tutorial.isl
client stubs for interface "Tutorial" to Tutorial.py ...
server stubs for interface "Tutorial" to Tutorial__skel.py ...
%
```

0.3.2 Building the Implementation

To provide an implementation of our interface, we *subclass* the generated Python class for our `Calculator` class:

```
# CalculatorImpl.py

import Tutorial, Tutorial__skel

class Calculator (Tutorial__skel.Calculator):

        def __init__ (self):
                self.the_value = 0.0

        def SetValue (self, v):
                self.the_value = v

        def GetValue (self):
                return self.the_value

        def Add (self, v):
                self.the_value = self.the_value + v

        def Subtract (self, v):
                self.the_value = self.the_value - v

        def Multiply (self, v):
                self.the_value = self.the_value * v

        def Divide (self, v):
                try:
                        self.the_value = self.the_value / v
                except ZeroDivisionError:
                        raise Tutorial.DivideByZero
```

Each instance of a `CalculatorImpl.Calculator` object inherits from `Tutorial__skel.Calculator`, which in turn inherits from `Tutorial.Calculator`. Each has an instance variable called `the_value`, which maintains a running total of the 'accumulator' for that instance. We can create an instance of a `Tutorial.Calculator` object by simply calling `CalculatorImpl.Calculator()`.

So, a very simple program to use the `Tutorial` module might be the following:

```
# simple1.py, a simple program that demonstrates the use of the
#  Tutorial true module as a library.
#
# run this with the command "python simple1.py NUMBER [NUMBER...]"
#

import Tutorial, CalculatorImpl, string, sys

# A simple program:
#  1)  make an instance of Tutorial.Calculator
#  2)  add all the arguments by invoking the Add method
#  3)  print the resultant value.

def main (argv):

        c = CalculatorImpl.Calculator()
        if not c:
                error("Couldn't create calculator")

        # clear the calculator before using it

        c.SetValue (0.0)

        # now loop over the arguments, adding each in turn */

        for arg in argv[1:]:
                v = string.atof(arg)
                c.Add (v)

        # and print the result

        print "the sum is", c.GetValue()
        sys.exit(0)

main(sys.argv)
```

This program would be compiled and run as follows:

```
% python simple1.py 34.9 45.23111 12
the sum is 92.13111
%
```

This is a completely self-contained use of the `Tutorial` implementation; when a method is called, it is the true method that is invoked. The use of ILU in this program adds some overhead in terms of included code, but has almost the same performance as a version of this program that does not use ILU.

0.3.3 Checking for Exceptions

Suppose, instead of the `Add` method, we'd called the `Divide` method. In that case, we might have had to *handle* a `DivideByZero` exception; that is, notice the exception and do something sensible. We do this by establishing a handler for the exception:

```
...
# now loop over the arguments, Dividing by each in turn */

try:
        for arg in argv[2:]:
                v = string.atof(arg)
                c.Divide (v)
except:
        print 'exception signalled:  ' + str(sys.exc_type)
        sys.exit(1)
...
```

And here's an example of what we get when it runs:

```
% python simple2.py 12345 6 7 8 9
the sum is 4.08234126984
% python simple2.py 12345 6 0 8 9
exception signalled:  Tutorial: DivideByZero
```

Actually, every method may return an exception, as there are a number of standard system exceptions which may be signalled even by methods which have no declared exceptions. So we should check every method to see if it succeeded, even simple ones like `GetValue`.

0.4 Providing the True Module as a Network Service

Now let's see what's involved in providing the calculator functionality as a network service. Basically, there are three things to look at:

- providing a "factory" to build calculator objects;
- publishing the name of the factory; and
- writing a server program.

0.4.1 Using Factories to Build Objects

When one program uses code from another address space, it has to get its hands on an instance of an ILU object, to be able to call methods. In our library application, we simply made a call into the true module, to create an instance of the calculator object. In the networked world, we need to do the same kind of thing, but this time the call into the true module has to be a method on an object type. In short, we need to have some object type which exports a method something like

```
CreateCalculator () : Calculator
```

There are several ways to provide this. The standard way of doing it is to add an object type to our `Tutorial` interface, which contains this method. This kind of object type is sometimes called a *factory*, because it exists only in order to build instances of other object types. We'll add the following type definition to our '`Tutorial.isl`':

```
TYPE Factory = OBJECT
  METHODS
    CreateCalculator () : Calculator
  END;
```

Then we need to provide an implementation of the `Factory` object type, just as we did with the `Calculator` type:

```
import Tutorial, Tutorial__skel, CalculatorImpl

class Factory (Tutorial__skel.Factory):

        # have the __init__ method take handle and server args
        # so that we can control which ILU kernel server is used,
        # and what the instance handle of the Factory object on
        # that server is.  This allows us to control the object ID
        # of the new Factory instance.

        def __init__(self, handle=None, server=None):
                self.IluInstHandle = handle
                self.IluServer = server

        def CreateCalculator (self):
                return (CalculatorImpl.Calculator())
```

Now, to provide other programs a way of creating calculator objects, we'll just create just one instance of `Tutorial.Factory`, and let programs call the `CreateCalculator` method on that at will, to obtain new calculator objects.

0.4.2 Publishing a Well-Known Instance

The question then arises, how does a program that wants to use the `Factory` object get its hands on that one well-known instance? The answer is to use the *simple binding* system built into ILU. Simple binding allows a program acting as a "server" to *publish* the location of a well-known object, and allows programs acting as "clients" of that server to look up the location, given the object's name.

The name of an ILU object instance has two parts, which are the *instance handle* of the object, and the name of its *kernel server*, called the *server ID*. (The kernel server is a data structure maintained by the ILU kernel which takes care of all communication between different address spaces.) These two combined must form a universally unique ID for the object. Usually you can simply let the ILU system choose names for your objects automatically, in which case it takes care to choose names which will never conflict with names in use by others. However, for objects which we wish to publish, we need to specify what the name of an object will be, so that users of the well-known object can find it.

When working with the Python programming language, this act of explicitly specifying an object name is divided into two steps. First, we create a kernel server with a specified server ID. Secondly, we create an instance of an object on this new server, with a specified instance handle. Together, the server ID and the instance handle form the name of the instance.

For instance, we might use a server ID of `Tutorial.`*domain*, where *domain* is your Internet domain (typically something like *department*`.`*company*`.com`, or *department*`.`*univerity*`.edu`). This serves to distinguish your server from other servers on the net. Then we can use a simple instance handle, like `theFactory`. The name, or *object ID*, of this object would then be `theFactory@Tutorial.`*domain*, where *domain* would vary from place to place. Note that this implies that only one instance of this object is going to exist in the whole domain. If you have many people using different versions of this object in your domain, you should introduce more qualifiers in the server ID so that your kernel server can be distinguished from that run by others.

0.4.3 The Server Program

Given this information, we can now write a complete program that will serve as a provider of calculator objects to other programs. It will create a single `Factory` instance with a well-known name, publish that instance, then hang out servicing methods invoked on its objects. Here's what it looks like:

```
# server.py -- a program that runs a Tutorial.Calculator server
#

import ilu, FactoryImpl, sys

def main(argv):

        if (len(argv) < 2):
                print "Usage:  python server.py SERVER-ID"
                sys.exit(1)

        # Create a kernel server with appropriate server ID, which
        #  is passed in as the first argument

        theServer = ilu.CreateServer (argv[1])

        # Now create an instance of a Factory object on that server,
        #  with the instance handle "theFactory"

        theFactory = FactoryImpl.Factory ("theFactory", theServer)

        # Now make the Factory object "well-known" by publishing it.

        theFactory.IluPublish()
```

```
        # Now we print the string binding handle (the object's name plus
        # its location) of the new instance.

        print "Factory instance published."
        print "Its SBH is '" + theFactory.IluSBH() + "'."

        handle = ilu.CreateLoopHandle()
        ilu.RunMainLoop (handle)


main(sys.argv)
```

When we run this program, we'll see something like:

```
% python server.py Tutorial.dept.company.com &
Factory instance published.
Its SBH is 'theFactory@Tutorial.dept.company.com@somegibberish'.
%
```

This indicates that the object known as `theFactory@Tutorial.`*dept.company.com* is being exported in a particular way, which is encoded in the *somegibberish* part of the string binding handle. Your specific numbers will vary, but it should look similar.

0.5 Using the Network Service

Given that someone has exported a module as a network service, by publishing the location of a well-known instance of an object type, a potential client of that module can then use the module by binding to that well-known instance. It does this by calling the standard ILU routine `ilu.LookupObject()`, which takes the name and type of an instance, and attempts to find that instance on the net. The name of the object is specified as a pair of strings, the *server ID* of the object's *kernel server*, and the *instance handle* of the object on that kernel server.

So, in our first example, we could replace the call to `Create_Tutorial_Calculator` with a routine that calls `ilu.LookupObject()` to find the factory, then creates an instance of a `Calculator`. The full code of the revised example, '`simple3.py`', is available as Appendix H [simple3.py], page 31, but here's what the new code for obtaining an instance of a `Calculator` looks like:

```
def Get_Tutorial_Calculator (factoryObjectSID, factoryObjectIH):

        # We have to call ilu.LookupObject() with the object ID of
        # the factory object, and the ``type'' of the object we're looking
        # for, which is always available as MODULE.TYPENAME

        f = ilu.LookupObject (factoryObjectSID, factoryObjectIH, Tutorial.Factory)
        if not f:
                print "Can't find Tutorial.Factory instance " + factoryObject-
SID + factoryObjectIH
                sys.exit(1)
        c = f.CreateCalculator()
        return (c)
```

We then can use the `simple3` program:

```
% python simple3.py Tutorial.dept.company.com theFactory 1 2 3 4 5 6
the sum is 2.10000
%
```

0.6 Subtyping and Other ISL Types

ILU ISL contains support for a number of types other than object types and `REAL`. The primitive ISL types include 16, 32, and 64 bit signed and unsigned integers, bytes, 8 and 16 bit characters, a boolean type, and 32, 64, and 128 bit floating point types. A number of type constructors allow specification of arrays, sequences, records, unions, and enumerations, as well as object types. The ISL `OPTIONAL` type constructor provides an implicit union of some type with `NULL`, which is useful for building recursive data structures such as linked lists or binary trees.

To illustrate some of these types, we'll extend the `Tutorial.Calculator` type. Many real-world desktop calculators include a *register tape*, a printed listing of all the operations that have been performed, with a display of what the value of the calculator was after each operation. We'll add a register tape to `Tutorial.Calculator`.

We could do it by adding a new method to `Tutorial.Calculator`, called `GetTape`. Unfortunately, this would break our existing code, because it would change the `Tutorial.Calculator` object type, and existing compiled clients wouldn't be able to recognize the new object type. Instead, we'll extend the object type by *subtyping*; that is, by creating a new object type which uses `Tutorial.Calculator` as a *supertype*, but adds new methods of its own. This subtype will actually have two types; both its own new type, and `Tutorial.Calculator`. We'll also define a subtype of the `Tutorial.Factory` type, to allow us to create new instances of the new `Calculator` subtype. Finally, we'll define a new module interface for the new types, so that we don't have to modify the `Tutorial` interface.

First, let's define the necessary type to represent the operations performed on the calculator:

```
INTERFACE Tutorial2 IMPORTS Tutorial END;

TYPE OpType = ENUMERATION
    SetValue, Add, Subtract, Multiply, Divide END;

TYPE Operation = RECORD
    op : OpType,
    value : REAL,
    accumulator : REAL
  END;

TYPE RegisterTape = SEQUENCE OF Operation;
```

The enumerated type `OpType` defines an abstract type with five possible values. The type `Operation` defines a record type (in Python, a dictionary) with 3 fields: the *op* field, which tells us which of the five possible calculator operations was performed, the *value* field, which tells us the value of the operand for the operation, and the *accumulator* field, which tells us what the value of the calculator was after the operation had been performed. Finally, the `Operation` type is a simple sequence, or list, of `Operation`. Note that `Tutorial2` *imports* `Tutorial`; that is, it allows the use of the `Tutorial` types, exceptions, and constants, in the specifications in `Tutorial2`.

Now we define the new object types (in the same file):

```
TYPE TapeCalculator = OBJECT COLLECTIBLE
  SUPERTYPES Tutorial.Calculator END
  DOCUMENTATION "4 function calculator with register tape"
  METHODS
    GetTape () : RegisterTape
  END;

TYPE Factory = OBJECT SUPERTYPES Tutorial.Factory END
  METHODS
    CreateTapeCalculator () : TapeCalculator
  END;
```

The `SUPERTYPES` attribute of an object type may take multiple object type names, so ISL supports multiple inheritance. The `Tutorial2.TapeCalculator` type will now support the six methods of `Tutorial.Calculator`, as well as its own method, `GetTape`.

We then need to provide an implementation for `Tutorial2.TapeCalculator` (see Appendix J [TapeCalculatorImpl.py], page 34 for the actual code). We modify each method on the `TapeCalculator` object to record its invocation, and add a slot to hold the contents of the 'tape'.

We also provide an implementation for `Tutorial2.Factory`:

```
import Tutorial2, Tutorial2__skel, TapeCalculatorImpl

class Factory (Tutorial2__skel.Factory):

        # have the __init__ method take handle and server args
        # so that we can control which ILU kernel server is used,
        # and what the instance handle of the Factory object on
        # that server is.  This allows us to control the object ID
        # of the new Factory instance.

        def __init__(self, handle=None, server=None):
                self.IluInstHandle = handle
                self.IluServer = server

        def CreateCalculator (self):
                return (TapeCalculatorImpl.TapeCalculator())

        CreateTapeCalculator = CreateCalculator
```

Note that both the `Tutorial2.Factory.CreateCalculator` and `Tutorial2.Factory.CreateTapeCalculator` methods create and return instances of `Tutorial2.TapeCalculator`. This is valid, because instances of `Tutorial2.TapeCalculator` are also instances of `Tutorial.Calculator`.

Now we modify '`server.py`' to create an instance of `Tutorial2.Factory`, instead of `Tutorial.Factory`, and to initialize the `Tutorial2` true-side code (See Appendix L [server2.py], page 36 for the actual code).

Finally, see Appendix M [simple4.py], page 37, for an example of a client of the `TapeCalculator` type.

Note that one nice result of this approach to versioning is that old clients, which know nothing about the new `TapeCalculator` class, or about the whole `Tutorial2` interface in general, will continue to function, since every instance of `Tutorial2.TapeCalculator` is also an instance of `Tutorial.Calculator`, and every instance of `Tutorial2.Factory` is also an instance of `Tutorial.Factory`.

0.7 Using OMG IDL with ILU

ILU also supports the use of the interface definition language OMG IDL, defined by the Object Management Group (OMG) for their Common Object Request Broker Architecture (CORBA). OMG IDL uses a C++-like syntax, so it may be easier for Python and C++ programmers to use than ILU ISL. Unfortunately, CORBA doesn't include some of the concepts in ILU, such as garbage collection for transient objects, or `OPTIONAL` types, so not every ILU interface can be expressed in OMG IDL, but many of them can. For example, here is the OMG IDL version of the `Tutorial` interface:

```
module Tutorial {

  exception DivideByZero {};

  interface Calculator {
    // Set the value of the calculator to ‘v’
    void SetValue (in double v);
    // Return the value of the calculator
    double GetValue ();
    // Adds ‘v’ to the calculator’s value
    void Add (in double v);
    // Subtracts ‘v’ from the calculator’s value
    void Subtract (in double v);
    // Multiplies the calculator’s value by ‘v’
    void Multiply (in double v);
    // Divides the calculator’s value by ‘v’
    void Divide (in double v) raises (DivideByZero);
  };

  interface Factory {
    // Create and return an instance of a Calculator object
    Calculator CreateCalculator();
  };
};
```

This can be used with the `python-stubber`:

```
% python-stubber Tutorial.idl
client stubs for interface "Tutorial" to Tutorial.py ...
server stubs for interface "Tutorial" to Tutorial__skel.py ...
%
```

This will be a bit slower than running the `python-stubber` on the equivalent ISL file, as the program works by converting the OMG IDL into ISL, then compiling from the ISL description. OMG IDL interfaces can be checked by running the OMG IDL-to-ILU ISL converter, `idl2isl`, directly:

```
% idl2isl Tutorial.idl
INTERFACE Tutorial;

EXCEPTION DivideByZero;
TYPE Calculator = OBJECT OPTIONAL
        METHODS
                SetValue (v : REAL),
                GetValue () : REAL,
                Add (v : REAL),
                Subtract (v : REAL),
                Multiply (v : REAL),
                Divide (v : REAL)
                        RAISES DivideByZero END
        END;
TYPE Factory = OBJECT OPTIONAL
        METHODS
                CreateCalculator () : Calculator
        END;
%
```

You will notice that the ISL interface generated by `idl2isl` is a bit different, in that the object type modifier `OPTIONAL` is used in the description of the `Calculator` and `Factory` types. This is because CORBA has the notion that any object type instance passed as a parameter or return value (or field in an array, or element of a sequence, etc.) may be `None`, instead of being a valid instance pointer. Thus, when working with OMG IDL descriptions of your interfaces, it is necessary to check the return type of methods like `Tutorial.Factory.CreateCalculator` to see that a valid object reference has been returned, before using the object. ISL allows you to have these CORBA-style objects, by using the `OPTIONAL` modifier in the declaration of an object type, but it also allows object pointers which can't be `None`. By default ILU object instances may not be `None`.

The OMG IDL version of `Tutorial2` can be found in Appendix O [Tutorial2.idl], page 40.

0.8 Transient and Permanent Objects

Keen-eyed readers will have seen that the `Calculator` type in '`Tutorial.isl`' is marked with the modifier `COLLECTIBLE`, while the `Factory` type is not. ILU differentiates between two kinds of objects, which we call *transient* and *permanent*.

Transient objects are typically created for a specific use, and typically have a fixed lifetime. They often have state associated with them. They are frequently used to represent some ongoing computation or operation.

Permanent objects are usually expected to be around forever (or until some activity not captured by a programming concept terminates). Permanent objects are often gateways to some sort of service, or represent some external piece of data, such as a cell in a spreadsheet, or a file on a disk somewhere. They are often stateless; when they do have state, it is typically backed by some form of stable storage, as they have to be available across crashes of their server program.

ILU provides an automatic system for *distributed garbage collection* of transient objects; that is, of removing them from everyone's address space when everyone is done with them. This uses the underlying garbage collection techniques of the particular programming language being used. With Python, the ILU garbage collector works in conjunction with the Python collector; with ANSI C, we use the extremely primitive collector provided for C; that is, when the process terminates, the objects are collected. To indicate to ILU that a particular object type is to be treated as transient, you mark the object type specification, in the ISL, with the keyword `COLLECTIBLE`.

For permanent objects, like the `Tutorial.Factory` object type, the ILU Python support will maintain a reference to the instance, preventing the Python collector from collecting it.

0.9 Using ILU with Tkinter

Python comes with a module called `Tkinter`, which is an interface to John Ousterhout's Tk user interface system. When you use both ILU and Tkinter together in a program, you should call the `ilu_tk` main loop to block and dispatch requests, instead of the `ilu` main loop. To illustrate this, here is a version of the `server2.py` program which puts up a Tk button. When the user presses the button, the server exits. Notice that instead of calling `ilu.RunMainLoop()` to block indefinitely, the program calls `ilu_tk.RunMainLoop()`. This call allows both Tk and ILU events to be handled when neither a Tk callback nor an ILU method call is active.

```
# server3.py -- a program that runs a Tutorial.Calculator server
#  Puts up a Tk button to kill it with.

# import the ilu_tk module first, so that
# initialization happens in the right order:
import ilu_tk

# now import the other modules...
import ilu, FactoryImpl2, sys, Tkinter

def main(argv):

        def quit():
                sys.exit(0)

        if (len(argv) < 2):
                print "Usage:  python server3.py SERVER-ID"
                sys.exit(1)

        theServer = ilu.CreateServer (argv[1])
        theFactory = FactoryImpl2.Factory ("theFactory", theServer)
        theFactory.IluPublish()

        # Now we put up a Tk button so that the user can kill the
        #  server by pressing the button

        f = Tkinter.Frame() ; Tkinter.Pack.config(f)
        b = Tkinter.Button (f, {'text' : theFactory.IluObjectID(),\
                                'command': quit})
        b.pack ({'side': 'left', 'fill': 'both'})

        # Then we wait in the ilu_tk mainloop, instead of either
        #  the ILU mainloop or the Tkinter mainloop

        ilu_tk.RunMainLoop()
```

```
main(sys.argv)
```

Appendix A General ILU Info

The Inter-Language Unification system (ILU) is a multi-language object interface system. The object interfaces provided by ILU hide implementation distinctions between different languages, between different address spaces, and between operating system types. ILU can be used to build multi-lingual object-oriented libraries ("class libraries") with well-specified language-independent interfaces. It can also be used to implement distributed systems. It can also be used to define and document interfaces between the modules of non-distributed programs.

The 2.0 release of ILU contains support for the programming languages ANSI C, C++, Modula-3, Python, and Common Lisp. It has been installed on many flavors of UNIX, including SPARC machines running SunOS 4.1.3 and Solaris 2, SGI MIPS machines running IRIX 5.2, Intel 486 machines running Linux 1.1.78, DEC Alpha machines with OSF/1, IBM RS/6000 machines running AIX, and HP machines running HP/UX. It runs on Microsoft Windows 3.1, Windows 95, and Windows NT environments. It supports both threaded and non-threaded operation. Since one of the implementation goals of ILU is to maximize compatibility with existing open standards, ILU provides support for use of the OMG CORBA IDL interface description language, and can be thought of as a CORBA ORB system (though with omissions from and extensions to the CORBA spec). As another result, ILU includes a self-contained implementation of ONC RPC.

ILU is available free from `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

Appendix B Tutorial.isl

```
INTERFACE Tutorial;

EXCEPTION DivideByZero
  "this error is signalled if the client of the Calculator calls
the Divide method with a value of 0";

TYPE Calculator = OBJECT COLLECTIBLE
  DOCUMENTATION "4-function calculator"
  METHODS
    SetValue (v : REAL) "Set the value of the calculator to ‘v’",
    GetValue () : REAL  "Return the value of the calculator",
    Add (v : REAL)      "Adds ‘v’ to the calculator’s value",
    Subtract (v : REAL) "Subtracts ‘v’ from the calculator’s value",
    Multiply (v : REAL) "Multiplies the calculator’s value by ‘v’",
    Divide (v : REAL) RAISES DivideByZero END
      "Divides the calculator’s value by ‘v’"
  END;

TYPE Factory = OBJECT
  METHODS
    CreateCalculator () : Calculator
  END;
```

Appendix C CalculatorImpl.py

```
import Tutorial, Tutorial__skel

class Calculator (Tutorial__skel.Calculator):

        def __init__ (self):
                self.the_value = 0.0

        def SetValue (self, v):
                self.the_value = v

        def GetValue (self):
                return self.the_value

        def Add (self, v):
                self.the_value = self.the_value + v

        def Subtract (self, v):
                self.the_value = self.the_value - v

        def Multiply (self, v):
                self.the_value = self.the_value * v

        def Divide (self, v):
                try:
                        self.the_value = self.the_value / v
                except ZeroDivisionError:
                        raise Tutorial.DivideByZero
```

Appendix D FactoryImpl.py

```
import Tutorial, Tutorial__skel, CalculatorImpl

class Factory (Tutorial__skel.Factory):

        # have the __init__ method take handle and server args
        # so that we can control which ILU kernel server is used,
        # and what the instance handle of the Factory object on
        # that server is.  This allows us to control the object ID
        # of the new Factory instance.

        def __init__(self, handle=None, server=None):
                self.IluInstHandle = handle
                self.IluServer = server

        def CreateCalculator (self):
                return (CalculatorImpl.Calculator())
```

Appendix E simple1.py

```
# simple1.py, a simple program that demonstrates the use of the
#  Tutorial true module as a library.
#
# run this with the command "python simple1.py NUMBER [NUMBER...]"
#

import Tutorial, CalculatorImpl, string, sys

# A simple program:
#  1)  make an instance of Tutorial.Calculator
#  2)  add all the arguments by invoking the Add method
#  3)  print the resultant value.

def main (argv):

        c = CalculatorImpl.Calculator()
        if not c:
                error("Couldn't create calculator")

        # clear the calculator before using it

        c.SetValue (0.0)

        # now loop over the arguments, adding each in turn */

        for arg in argv[1:]:
                v = string.atof(arg)
                c.Add (v)

        # and print the result

        print "the sum is", c.GetValue()
        sys.exit(0)

main(sys.argv)
```

Appendix F simple2.py

```
# simple2.py, a simple program that demonstrates the use of the
#  Tutorial true module as a library.
#
# run this with the command "python simple1.py NUMBER [NUMBER...]"
#

import Tutorial, CalculatorImpl, string, sys

# A simple program:
#  1)  make an instance of Tutorial.Calculator
#  2)  add all the arguments by invoking the Add method
#  3)  print the resultant value.

def main (argv):

        c = CalculatorImpl.Calculator()
        if not c:
                error("Couldn't create calculator")

        # clear the calculator before using it

        if (len(sys.argv) < 2):
                c.SetValue (0.0)
        else:
                c.SetValue (string.atof(argv[1]))

        # now loop over the arguments, Dividing by each in turn */

        try:
                for arg in argv[2:]:
                        v = string.atof(arg)
                        c.Divide (v)
        except:
                print 'exception signalled:  ' + str(sys.exc_type)
                sys.exit(1)

        # and print the result

        print "the sum is", c.GetValue()
        sys.exit(0)

main(sys.argv)
```

Appendix G server.py

```
# server.py -- a program that runs a Tutorial.Calculator server
#

import ilu, FactoryImpl, sys

def main(argv):

        if (len(argv) < 2):
                print "Usage:  python server.py SERVER-ID"
                sys.exit(1)

        # Create a kernel server with appropriate server ID, which
        #  is passed in as the first argument

        theServer = ilu.CreateServer (argv[1])

        # Now create an instance of a Factory object on that server,
        #  with the instance handle "theFactory"

        theFactory = FactoryImpl.Factory ("theFactory", theServer)

        # Now make the Factory object "well-known" by publishing it.

        theFactory.IluPublish()

        # Now we print the string binding handle (the object's name plus
        # its location) of the new instance.

        print "Factory instance published."
        print "Its SBH is '" + theFactory.IluSBH() + "'"

        handle = ilu.CreateLoopHandle()
        ilu.RunMainLoop (handle)


main(sys.argv)
```

Appendix H simple3.py

```
# simple3.py -- a simple client program that finds the Calculator Factory,
#   creates a calculator, and adds up its arguments as before
#
# to run:  python simple4.py ARG [ARG...]

import Tutorial, ilu, sys, string

# We define a new routine, "Get_Tutorial_Calculator", which
# finds the tutorial factory, then creates a new Calculator
# object for us.

def Get_Tutorial_Calculator (factoryObjectID):

        # We have to call ilu.LookupObject() with the object ID of
        # the factory object, and the ``type'' of the object we're looking
        # for, which is always available as MODULE.TYPENAME

        f = ilu.LookupObject (factoryObjectID, Tutorial.Factory)
        if not f:
                print "Can't find Tutorial.Factory instance " + factoryObjectID
                sys.exit(1)
        c = f.CreateCalculator()
        return (c)

def main (argv):

        # A simple program:
        #  1)  make an instance of Tutorial.Calculator
        #  2)  add all the arguments by invoking the Add method
        #  3)  print the resultant value.

        if (len(argv) < 2):
                print "Usage:  python simple3.py FACTORY-OBJECT-ID NUMBER [NUMBER...]\n",
                sys.exit(1)

        c = Get_Tutorial_Calculator(argv[1])
        if not c:
                print "Couldn't create calculator"
                sys.exit(1)

        # clear the calculator before using it

        c.SetValue (0.0)

        # now loop over the arguments, adding each in turn

        for arg in argv[2:]:
```

```
                v = string.atof (arg)
                c.Add (v)

        # and print the result

        print "the sum is " + str(c.GetValue())

        sys.exit (0);


main(sys.argv)
```

Appendix I Tutorial2.isl

```
INTERFACE Tutorial2 IMPORTS Tutorial END;

TYPE OpType = ENUMERATION
    SetValue, Add, Subtract, Multiply, Divide END;

TYPE Operation = RECORD
    op : OpType,
    value : REAL,
    accumulator : REAL
  END;

TYPE RegisterTape = SEQUENCE OF Operation;

TYPE TapeCalculator = OBJECT COLLECTIBLE
  SUPERTYPES Tutorial.Calculator END
  DOCUMENTATION "4 function calculator with register tape"
  METHODS
    GetTape () : RegisterTape
  END;

TYPE Factory = OBJECT SUPERTYPES Tutorial.Factory END
  METHODS
    CreateTapeCalculator () : TapeCalculator
  END;
```

Appendix J TapeCalculatorImpl.py

```
import Tutorial, Tutorial2, Tutorial2__skel

class TapeCalculator (Tutorial2__skel.TapeCalculator):

        def __init__ (self):
                self.value = 0.0
                self.tape = []

        def SetValue (self, v):
                global value
                self.value = v
                self.tape.append({'op' : Tutorial2.OpType__SetValue, 'value'
: v, 'accumulator' : self.value})

        def GetValue (self):
                return self.value

        def Add (self, v):
                self.value = self.value + v
                self.tape.append({'op' : Tutorial2.OpType__Add, 'value' : v,
'accumulator' : self.value})


        def Subtract (self, v):
                self.value = self.value - v
                self.tape.append({'op' : Tutorial2.OpType__Subtract, 'value'
: v, 'accumulator' : self.value})


        def Multiply (self, v):
                self.value = self.value * v
                self.tape.append({'op' : Tutorial2.OpType__Multiply, 'value'
: v, 'accumulator' : self.value})


        def Divide (self, v):
                try:
                        self.value = self.value / v
                except ZeroDivisionError:
                        raise Tutorial.DivideByZero
                self.tape.append({'op' : Tutorial2.OpType__Divide, 'value' :
v, 'accumulator' : self.value})

        def GetTape (self):
                return (self.tape)
```

Appendix K FactoryImpl2.py

```
import Tutorial2, Tutorial2__skel, TapeCalculatorImpl

class Factory (Tutorial2__skel.Factory):

        # have the __init__ method take handle and server args
        # so that we can control which ILU kernel server is used,
        # and what the instance handle of the Factory object on
        # that server is.  This allows us to control the object ID
        # of the new Factory instance.

        def __init__(self, handle=None, server=None):
                self.IluInstHandle = handle
                self.IluServer = server

        def CreateCalculator (self):
                return (TapeCalculatorImpl.TapeCalculator())

        CreateTapeCalculator = CreateCalculator
```

Appendix L server2.py

```
# server2.py -- a program that runs a Tutorial2.TapeCalculator factory
#

import ilu, FactoryImpl2, sys

def main(argv):

        if (len(argv) < 2):
                print "Usage:  python server2.py SERVER-ID"
                sys.exit(1)

        theServer = ilu.CreateServer (argv[1])
        theFactory = FactoryImpl2.Factory ("theFactory", theServer)

        # Now make the Factory object "well-known" by publishing it.

        theFactory.IluPublish()

        # Now we print the string binding handle (the object's name plus
        # its location) of the new instance.

        print "Factory2 instance published."
        print "Its SBH is '" + theFactory.IluSBH() + "'"

        handle = ilu.CreateLoopHandle ();
        ilu.RunMainLoop (handle)


main(sys.argv)
```

Appendix M simple4.py

```
# simple4.py -- a simple client program that finds the TapeCalculator Factory,
#   creates a calculator, and provides a simple interactive calculator
#
# to run:  python simple4.py ARG [ARG...]

import Tutorial2, ilu, sys, string

# We define a new routine, "Get_Tutorial_Calculator", which
# finds the tutorial factory, then creates a new Calculator
# object for us.

def Get_Tutorial_Calculator (factoryObjectID):

  # We have to call ilu.LookupObject() with the object ID of
  # the factory object, and the ``type'' of the object we're looking
  # for, which is always available as MODULE.TYPENAME

  f = ilu.LookupObject (factoryObjectID, Tutorial2.Factory)
  if not f:
    print "Can't find Tutorial.Factory instance " + factoryObjectID
    sys.exit(1)
  c = f.CreateTapeCalculator()
  return (c)


opname = ['SetValue', 'Add', 'Subtract', 'Divide', 'Multiply']

def Print_Tape (tape):

  # print out the Calculator tape nicely

  for op in tape:
    print "  %s(%f) => %f" % (opname[op['op']], op['value'], op['accumulator'])

def main (argv):

  if (len(argv) < 2):
    print "Usage:  python simple3.py FACTORY-OBJECT-ID NUMBER [NUMBER...]\n",
    sys.exit(1)

  c = Get_Tutorial_Calculator(argv[1])
  if not c:
    print "Couldn't create calculator"
    sys.exit(1)

  # clear the calculator before using it
```

```
    newval = 0.0
    c.SetValue (newval)

    quitflag = 0

    while not quitflag:

      sys.stdout.write("%.5f\n> " % newval)
      sys.stdout.flush()

      line = sys.stdin.readline()

      if (not line):
        sys.exit(0)

      try:
        if (line[0] == '\n'):
          pass
        elif (line[0] == '+'):
          val = string.atof(line[1:-1])
          c.Add(val)
        elif (line[0] == '-'):
          val = string.atof(line[1:-1])
          c.Subtract(val)
        elif (line[0] == '*'):
          val = string.atof(line[1:-1])
          c.Multiply(val)
        elif (line[0] == '/'):
          val = string.atof(line[1:-1])
          c.Divide(val)
        elif (line[0] == 'q'):
          quitflag = 1
        elif (line[0] == 'c'):
          c.SetValue(0.0)
        elif (line[0] == 't'):
          Print_Tape ( c.GetTape() )
        else:
          print "Invalid operation <" + line[:-1] + ">"
          print "Valid ops are +, -, *, /, tape, clear, quit"

        newval = c.GetValue()

      except:
        print "Operation <%s> signals error <%s>." % (line[:-1], sys.exc_type)

    sys.exit(0)


  main(sys.argv)
```

Appendix N Tutorial.idl

```
module Tutorial {

  exception DivideByZero {};

  interface Calculator {

    // Set the value of the calculator to 'v'
    void SetValue (in double v);
    // Return the value of the calculator
    double GetValue ();
    // Adds 'v' to the calculator's value
    void Add (in double v);
    // Subtracts 'v' from the calculator's value
    void Subtract (in double v);
    // Multiplies the calculator's value by 'v'
    void Multiply (in double v);
    // Divides the calculator's value by 'v'
    void Divide (in double v) raises (DivideByZero);
  };

  interface Factory {

    // Create and return an instance of a Calculator object
    Calculator CreateCalculator();
  };
};
```

Appendix O Tutorial2.idl

```
#include "Tutorial.idl"

module Tutorial2 {

  enum OpType { SetValue, Add, Subtract, Multiply, Divide };

  struct Operation {
    OpType op;
    double value;
    double accumulator;
  };

  typedef sequence<Operation> RegisterTape;

  // A four function calculator with a register tape
  interface TapeCalculator : Tutorial::Calculator {

    RegisterTape GetTape ();
  };

  // A factory that produces TapeCalculators
  interface Factory : Tutorial::Factory {

    TapeCalculator CreateTapeCalculator ();
  };
};
```

Appendix P server3.py

```
# server3.py -- a program that runs a Tutorial.Calculator server
#  Puts up a Tk button to kill it with.

import ilu, FactoryImpl2, sys, Tkinter, ilu_tk

def main(argv):

        def quit():
                sys.exit(0)

        if (len(argv) < 2):
                print "Usage:  python server3.py SERVER-ID"
                sys.exit(1)

        theServer = ilu.CreateServer (argv[1])
        theFactory = FactoryImpl2.Factory ("theFactory", theServer)
        theFactory.IluPublish()

        # Now we put up a Tk button so that the user can kill the
        #  server by pressing the button

        f = Tkinter.Frame() ; Tkinter.Pack.config(f)
        b = Tkinter.Button (f, {'text' : theFactory.IluObjectID(),\
                                'command': quit})
        b.pack ({'side': 'left', 'fill': 'both'})

        # Then we wait in the ilu_tk mainloop, instead of either
        #  the ILU mainloop or the Tkinter mainloop

        ilu_tk.RunMainLoop()


main(sys.argv)
```

Index of Concepts