User-Centered Evolutionary Software Development Using Python and Java

Douglas Cunningham
Carnegie Mellon University

Abstract

The two language approach to software development has been investigated by several language designers. The primary hypothesis of such an approach being that both strong compile-time type checking and loose run-time type checking are desirable in evolutionary software development. Java is a strongly typed language which offers performance, robustness and modularity as such, while Python is a weakly typed language which offers rapid prototyping, dynamic run-time modification, and delayed evaluation. The premise of this work is that evolutionary software development using both languages together can be more efficient than using either language alone.

The Java Python Interface (JPI) is an interface between Java and Python which allows the two languages to interoperate through dynamic message lookup and the conversion and exchange of objects and exceptions. Through the JPI, Python be used as a scripting language for Java -- from a Python interpreter one can prototype AWT components and even create bindings which call Python code. In addition, the JPI can be used to add user level Python scripting to Java programs.

1 Introduction

Over the last decade, the use and value of prototypes in design, both in software and other engineering disciplines, has been investigated and discussed by researchers [Budde92] [Krogh96]. Evolutionary software development, which consists of the rapid prototyping of new components and their evolution into hardened components [Budde92], offers many benefits to the object-oriented software development process [Cox91], including the ability to address rapidly changing software requirements. In addition, the efficiency gained from user-centered development, which consists of constant user feedback impacting the software design, has been studied over the last few years [Landauer95].

The requirements of both a user-centered and evolutionary software development cycle are broad and inclusive of many diametric aspects. User-centered development demands the ability to rapidly add features, modify behavior, and create new user interface components. Evolutionary software development can have these same requirements but also requires the creation of well-defined modules and stable, well-performing code.

Clean module interfaces, compile-time error checking, and good performance make strongly typed, static languages such as C++ [Stroustrup87] or Java [Arnold96] useful for the creation of hardened components. Meanwhile, rapid prototyping, fast compile-test-debug cycles, and high programming flexibility make loosely typed, dynamic languages such as Python [Watters96] or Tcl [Ousterhout94] appropriate for the rapid creation of prototype components.

Researchers have claimed that a two language approach proves both feasible and practical for software development [Ousterhout97]. For example, languages such as Python and Tcl provide consistent interfaces to C to allow developers to move performance critical methods and procedures into C without modification of the calling code. In addition, the dynamic language can be used to make calls to and exchange data between existing C or C++ modules.

The Java Python Interface (JPI) attempts to merge the strengths of two languages which are very similar in syntax and function. The purpose of the JPI is two fold: 1) allow for the simultaneous development of prototype components in Python and hardened components in Java, and 2) allow for the addition of user level Python scripting to Java programs.

This paper will describe the motivations of such an approach, and then focus on the technical details of the JPI.

2 Motivation of the Two Language Approach

A series of pitfalls are associated with single language development situations which make the two language approach to software development very appealing.

If developers choose to use a strongly typed, static language they often encounter problems in the early prototyping and development phases. First, they may begin to define programming module interfaces prematurely which can cause delays for other developers when modules are redefined and require recompilation of part or all of the system. Second, they are forced to engage in long compile-test-debug cycles due to the longer compilation times of static languages and the need to restart the program and recreate the state under which the error occurred. Third, determining the cause of an error can be difficult due to the inability the directly query the running program for information other than the use of a debugger. Fourth, it is difficult to debug part of an incomplete program since the compiler will not allow the program to run until all of the compile-time errors are corrected. While a static language offers many benefits, these limitations often make it difficult to rapidly prototype all or part of an application.

On the other hand, if developers choose to use a weakly typed, dynamic language they often encounter problems in the latter phases of development when performance, scalability, and maintainability are crucial. First, they may end up not defining module interfaces well since a loosely typed language does not require the strict specification of the interface to a module. Second, they often encounter performance problems when writing computationally intensive code and as a result spend a great deal of time trying to optimize code. Third, they suffer scale problems related to speed and process size when a system becomes large. Fourth, the detection of all coding errors, except syntactical, is delayed until the actual line of code is executed. This means completely inclusive test cases must be written in order to check the validity of code. Despite the benefits of a highly dynamic language, these limitations make it extremely difficult to maintain a large piece of software.

Of course, alternatives exist to choosing just one of the above types of languages. Languages such as Python and Tcl, offer the option of using both a loosely typed, dynamic component, the language itself, and a strongly typed, static component, C. Therefore, a developer can use a language such a Python to prototype and debug new code and then reimplement that code in C for performance and scalability. This two language approach to software development offers many benefits over using a single language. Programmers can prototype new features and work out their logic and errors without the need to engage in a long compile-test-debug cycle. Once code logic has been established and verified, the code can then be moved into a strongly typed, static language to gain speed and robustness and force the creation of a well-defined interface.

For several years, the *n*-dim project at Carnegie Mellon University has also investigated the two language approach in an object system called BOS. BOS, standing for Basic Object System, allows for the prototyping of code in an interpreted language called stitch and the hardening of code in C, much like the Python and Tcl approach [Dutoit96]. Our experiences have shown that the main drawback of this approach is that moving code into C is a costly effort since it requires a complete reimplementation of the code and often the logic. As a result, developers often take the first step of prototyping code in the dynamic language and then neglect the needed step of reimplementing the code in the static language despite the performance and scalability gains. Recently, tools such as SWIG [Beazley97] make this process easier for languages such Perl, Python, and Tcl, but they still require significant work on the part of the programmer.

Python and Java share many common features [Masse96] — they are both object-oriented, support garbage collection, provide object-based exception handling, and have a similar syntax. As a result, using them together in the two language approach offers a potential solution to this problem of converting prototype code to hardened code. Given the ability to seamlessly communicate between the two languages, one could maintain different parts of a program in each language. Classes prototyped in Python could easily be converted to Java classes, but at the same time, particular methods, such as those undergoing a lot of rapid change, could be kept in Python. The similarities of Python and Java allow for the transition of code from one language to the other to be extremely easy and even offer the possibility of complete automation.

The Java Python Interface (JPI), provides a mechanism for this type of communication. Messages can be sent to Java objects from Python and vice-versa, argument conversions are performed between the two languages, objects from one language can ‘exist’ in the memory space of the other, and exceptions thrown from one language can be caught in the other. The following section describes the JPI in more detail and gives examples of how it can be used to satisfy the requirements of both rapid prototyping and code hardening in software development.

3 The JPI Overview

3.1 Description

The JPI is an interface between Java and Python. It allows for the dynamic manipulation of Java objects through the use of Python and vice-versa. By writing Python ‘scripts’ one can program using existing Java objects. In addition, the syntactical similarities between Python and Java are great enough that the conversion of prototyped Python code to hardened Java code is extremely easy and can even be largely automated.

A very simple example of the use of the JPI is the following Python program which creates a Java AWT button and prints ‘Hello World!’ when it is clicked.

```
# HelloWorld.py

import java
Frame = java.findClass("java/awt/Frame")
Button = java.findClass("java/awt/Button")
PyEventListener = java.findClass("PyEventListener")

class HelloWorld:

def __init__(this):
# Create a frame and a button
frame = Frame.new("Hello World Test")
button = Button.new("Click Here")
frame.add(button)
frame.pack()
frame.setVisible(1)

# Create a listener object which will call me back when
# an event occurs and add it to the button
listener = PyEventListener.new(this)
button.addActionListener(listener)
# Tell the listener what event to listen for and what message
# to send me when the event occurs
listener.listenFor("actionPerformed", "printMessage")

def printMessage(this, event):
print "Hello World!"

>>> from HelloWorld import *
>>> b = HelloWorld() # Button appears on screen - click away!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

This is done through the use of the Java Native Interface (JNI), three Java classes, and a Python C module. It is an extremely simple interface, but at the same time, it is quite powerful. It has already been used to prototype new interface components in Python and then move them into Java for speed and robustness. It has also been used to add user-level Python scripting to Java programs.

3.2 Implementation

The JPI consists of a Python C module and three simple Java classes. The Python C module named ‘java’ and a Java class named ‘Python’ implement the simple functionality of looking up classes and dynamically sending messages to Java objects. They also handle the conversion of objects between Java and Python with the addition of the Java class named ‘PyObject’. In addition, the ‘PyEventListener’ Java class is provided to listen for AWT events.

3.2.1 Message Sending and Field Accesses

Messages are sent back and forth between Java and Python through dynamic message lookup. The Python C module implements an object type, JavaObject, which wraps an actual java object. That is, the JavaObject contains a pointer to the actual Java object and when a message is sent to the Python JavaObject a dynamic message lookup ensues in order to find out if the actual Java object can answer the message. If it can not, an error is thrown. Likewise, the Java class, PyObject, wraps an actual Python object. An instance of PyObject contains a pointer to the actual Python object and a ‘send’ method is provided to send a message to the actual Python object. For example, the following code sends the ‘size’ message to a vector ‘v’:

```
>>> v.size()
```

The code to send a message to Python is not as clean. This is because Java does not provide any mechanism for answering any message sent to an object such as ‘getattr’. Here is an example piece of Java code to send ‘reverse’ to the Python range ‘r’:

```
// Somewhere inside a Java method...
r.send(“reverse”);
```

Accessing a field of a Java object is done in much the same way. When a message is sent to a Java object from Python, the dynamic message lookup checks to see if a field access would also satisfy the name and arguments requirements of the message. If so, it will execute the field access, either get or set, but this is done only after no matching method can be found. Unfortunately, the accessing of fields is not as seamless as sending messages. For example, to print the contents of field ‘field’ on object ‘o’ you must do the following:

```
>>> print o.field()
```

This reason for this is that the JavaObject type uses the ‘getattr’ function to answer any messages sent to the object. Since the ‘getattr’ function is not given the information of whether the user is attempting to invoke a method or access a field, it is not possible to know whether a field lookup should be done. Presumably this information is available to Python, but it is not passed to ‘getattr’. An alternative solution to forcing the use of parentheses is to have a field lookup take precedence over a method lookup, but the behavior would still not be ideal and some odd error cases would exist.

3.2.2 Object Conversions

Sending messages without arguments or without receiving results would be extremely limiting. So in addition to sending a message, the JPI can also pass objects through type conversion. The supported type conversions are listed below in Table 1.

Table 1: Java to Python and Python to Java Conversions

Java Type	Python Type		Python Type	Java Type
Integer, int	int		int	Integer, int, Boolean, boolean
Long, long	long		long	Long, long
Float, float	float		float	Double, double, Float, float
Double, double	float		str	String
Boolean, boolean	0, 1		object	PyObject
String	str		JavaObject	Object
PyObject	object		None	null
Object	JavaObject			
null	None			

The argument conversion code will convert objects in both directions when used in an argument list or when returning results. If an object can not be convert to a primitive type it will be converted to either a PyObject or a JavaObject depending on the direction of conversion. For example, the following code contains several object conversions.

```
# Java Vector class and new instance are converted to JavaObject type
>>> v = java.findClass(“java/util/Vector”).new()
# No conversions here
>>> r = range(5)
# Python range is converted to a PyObject
>>> v.addElement(r)
# int 0 is converted to Java int 0 and element found is converted
```

```
# from PyObject back to Python range
>>> print v.elementAt(0)
[0, 1, 2, 3, 4]
>>>
```

The JPI attempts to support object conversions well in both directions, but there are no guarantees that a problematic situation can not arise. For example, if there existed a Java class as below:

```
public class Test() {
public void set(int i) { }
public void set(boolean b) { }
}
```

Then the argument conversion code must make a choice when the following Python code is executed:

```
>>> o = java.findClass("Test").new()
>>> o.set(1)
```

The choice is currently arbitrary — the first 'set' method it finds will be the one selected. There is, of course a way around this. An Integer or Boolean object can be created and sent as the argument instead. For example, the following code creates an Integer and forces the explicit invocation of the 'set' method taking an int argument:

```
>>> o = java.findClass("Test").new()
>>> Integer = java.findClass("java/lang/Integer")
>>> o.set(Integer.new(1))
```

3.2.3 Exception Handling

Exceptions are handled by the JPI by converting an exception back and forth between a Python exception and a Java exception as it passes up the call stack. The programmer can catch the exception in either of the languages.

Currently though, the exception handling mechanism loses information about the original exception when it is converted between languages. So an exception occurring when sending a message to Java will result in a general Exception object being thrown. Due to this drawback, the JPI prints out the exception information just before conversion so that the programmer can see the actual exception that occurred. Future work will improve upon this such that in either language the actual exception which was thrown will be passed back and forth. For example, the current behavior is as follows:

```
>>> v = java.findClass("java/util/Vector").new()
>>> v.elementAt(0)
```

```
java.lang.ArrayIndexOutOfBoundsException: 0 >= 0
at java.util.Vector.elementAt(Vector.java)
at ndim.util.Python.dynamicMessageSend(Python.java:235)
at ndim.util.Python.readInput(Python.java:439)
at ndim.modeling.Test.main(Test.java:36)
Traceback (innermost last):
File "<string>", line 1, in ?
java.error: Exception thrown from Java
java.lang.Exception: Exception thrown from Python
at ndim.util.Python.readInput(Python.java:439)
at ndim.modeling.Test.main(Test.java:36)
>>>
```

While the future behavior will be:

```
>>> v = java.findClass("java/util/Vector").new()
>>> v.elementAt(0)
java.lang.ArrayIndexOutOfBoundsException: 0 >= 0
at java.util.Vector.elementAt(Vector.java)
at ndim.util.Python.dynamicMessageSend(Python.java:235)
at ndim.util.Python.readInput(Python.java:439)
at ndim.modeling.Test.main(Test.java:36)
>>>
```

3.2.4 Other Implementation Details

Working with Java Classes. To create a Java instance one must first get a class to instantiate. This is done using the 'findClass' method of the java module. It is useful to assign classes to variables for more convenient use. Below is an example.

```
>>> Frame = java.findClass("java/awt/Frame")
```

Once a class is obtained, the 'new' message will invoke a constructor matching the arguments provided. The following code will create a new frame with the title "My Frame":

```
>>> f = Frame.new("My Frame")
```

This is equivalent to the Java code:

```
new Frame("My Frame");
```

To access static fields or methods, a message can be sent to the class:

```
>>> Color = java.findClass("java/awt/Color")
>>> Color.black()
```

Interfaces and the PyEventListener class. In order to have a Python class implement a Java interface, a Java class implementing the interface must be created which then forwards messages to a Python object. The PyEventListener class is an example of this. The PyEventListener class implements all of the AWT event listener interfaces. Its constructor requires a PyObject to which it will forward the interface methods. The PyEventListener serves merely as a custom wrapper for a PyObject to satisfy the interface requirements of methods which demand some type of event listener. The following code shows how the PyEventListener can be used:

```
# This code assumes a Python object 'o' exists

# A button 'b' is created
>>> b = Button.new("click here")
# An event listener 'l' is created which will forward messages to 'o'
>>> l = java.findClass("PyEventListener").new(o)
# 'l' can be added as an actionListener to 'b' since PyEventListener
# implements the ActionListener interface
>>> b.addActionListener(l)
# 'l' is configured to forward the 'actionPerformed' method to 'o'
>>> l.listenFor("actionPerformed", "actionPerformed")
```

The (Fake) Interpreter. The JPI provides its own interpreter because difficulties were encountered when trying to run the Python interpreter from C. This fake interpreter merely reads input and sends it to Python for the real interpretation, but as a result, the behavior of the interpreter does not exactly match that of Python. Multi-line expressions are finished as soon as a blank line is entered, and results are not printed, the user must print them manually (this will be fixed soon). For example, sending a message may seemingly do nothing:

```
>>> f.message() # returns "It happened!"
>>>
```

So instead, the programmer must do:

```
>>> print f.message()
It happened!
>>>
```

Reserved Words. There is also a problem in the sense that the Python parser looks for certain reserved words and generates an error if the reserved work is not being used properly. Therefore, trying to send a message to a Java object which matches a reserved word will result in a syntax error. For example, the following code yields a syntax error:

```
>>> o.print()
File "<string>", line 1
o.print()
```

```
^
SyntaxError: invalid syntax
java.lang.Exception: Exception thrown from Python
at ndim.util.Python.readInput(Python.java:439)
at ndim.modeling.Test.main(Test.java:36)
>>>
```

There is around for this. Instead, the following code can be used:

```
>>> o.send("print")
```

4 Conclusions

The JPI has been used extensively by the author. Despite the described shortcomings it has proven extremely valuable. Code can be prototyped very quickly in Python and, once it works properly, moved to Java. Programs can be interactively debugged such that the state of a running program can be queried and modified to find the cause of a bug and the potential corrections for it. Graphical interfaces can be prototyped using the Java AWT and modifications can be made directly from the Python command line such that layout and appearance can be experimented with without needing to recompile and rerun. User level scripts can be added to programs such that run-time customizations can be made to programs.

Due to the great similarities between Java and Python, the JPI is able to support the two language approach to software development extremely well. For example, in one particular case a LoginDialog class was prototyped in Python while establishing the look and feel and then reimplemented in Java once completed. The dynamic nature of Python allowed for many iterations of the look and feel to be experimented with over the course of a day. Once the final design was determined, the LoginDialog class was reimplemented in Java. The reimplementation consisted primarily of a syntax conversion and was completed in less than an hour.

5 References

[Arnold96] Arnold, K., J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[Beazley97] Beazley, D., *The SWIG User's Manual*, 1997.

[Budde92] Budde, R., K. Kautz, K. Kuhlenkamp, and H. Züllighoven, *Prototyping: An Approach to Evolutionary System Development*, Springer Verlag, New York, 1992.

[Cox91] Cox, B., *Object-Oriented Programming: an Evolutionary Approach*, Reading, Mass, 1991.

[Dutoit96] Dutoit, A., S. Levy, D. Cunningham, and R. Patrick, "The Basic Object System: Supporting a Spectrum From Prototypes to Hardened Code," *OOPSLA 96 Conference Proceedings*, Vol. 31, No. 10, pp. 104-121, October 1996.

[Krogh96] Krogh, B., S. Levy, A. Dutoit, and E. Subrahmanian, "Strictly Class-Based Modeling Considered Harmful," *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS)*, Maui, Hawaii, January 1996.

[Landauer95] Landauer, T. K., *The Trouble with Computers: Usefulness, Usability, and Productivity*, MIT Press, Cambridge, Mass., 1995.

[Masse96] Masse, R., "An Analysis of Two Next-Generation Languages: Java and Python," Unpublished paper (http://www.python.org/~rmasse/papers/java-python/), 1996.

[Ousterhout94] Ousterhout, J., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[Ousterhout97] Ousterhout, J., "Scripting: Higher Level Programming for the 21st Century," Unpublished draft (http://www.sunlabs.com/~ouster/scripting.html), Sun Microsystems Laboratories, 1997.

[Stroustrup87] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1987.

[Watters96] Watters, A., G. van Rossum, and J. C. Ahlstrom, *Internet Programming with Python*, MIS Press/Henry Holt Publishers, 1996.