An Agnostic Approach to Scripting Languages

Benjamin B. Bederson, James D. Hollan, Jason Stewart
Computer Science Department
University of New Mexico
Albuquerque, NM 87131
([*bederson, hollan, jasons*]@*cs.unm.edu*)
http://www.cs.unm.edu/pad++

KEYWORDS
Interactive user interfaces, multiscale zoomable interfaces, information visualization, information physics, interpreted scripting languages.

ABSTRACT
We describe a generic approach to connect any number of interpreted scripting languages to a library. We implemented this technique with Pad++ so that complete access to Pad++ is now available through Scheme and Perl as well as Tcl/Tk.

INTRODUCTION
Ask any two programmers what their favorite computer languages are and you will likely end up with a very intense debate. Indeed, developers' feelings about programming languages are almost religious in nature. With the current proliferation of scripting languages (Scheme, Tcl/Tk, Python, Perl, etc.), this divisiveness can be difficult for development teams.

In addition to strong personal preferences, there are many practical reasons to choose one language over another. Speed of execution, speed of coding, legacy code, portability, programmer familiarity with languages, availability of libraries, and appropriateness to application domain are all important factors in deciding which language to use.

For library developers such as ourselves, the huge number of available scripting languages is a real concern. We are developing Pad++ [1][2][3], a general-purpose substrate for supporting zoomable graphical interfaces. While our library is written in C++, we want to provide a scripting language interface so applications can be written at a higher level, and prototyped more easily. Until recently, we supported only Tcl/Tk [4] which was adequate, but limited the number of potential users of our system.

We now continue to support Tcl/Tk, but in addition, we provide a generic mechanism to add support for any other scripting language. We have used this mechanism to add support for Scheme (the ELK distribution [5]) and Perl [6]. Our approach provides these features:

- An arbitrary number of languages can be active in the same environment simultaneously.
- Source files can be loaded.
- A top-level interpreter is available for appropriate languages (see below) with the ability to switch between top-level languages at run-time.
- Callbacks, such as event bindings or timer scripts, can be written in any language.
- A single Makefile flag for each language determines at compile-time which languages will be supported at run-time.

There have been other approaches to the multiple language problem. STk provides a Scheme interface to Tk, but eliminates access to Tcl [7]. Guile provides a base language which is intended to provide the ability to write interpreters for any other scripting language [8]. This approach is promising, but is still under development and has the potential problem that each scripting language will be slower than the original since it will be interpreted in Guile rather than in its original implementation. Despite the quest for a universal language, it is unlikely that a single language will ever satisy everyones needs, and the issue of giving access to a library through as many languages as possible remains.

IMPLEMENTATION
Our approach to supporting multiple interpreted scripting languages rests on the fact that an entire language can be accessed through just a few simple functions. We distinguish between languages that only support loading of entire files (such as Java or Perl) with languages that support a top-level interpreter (such as Tcl or Scheme). For our purposes, all languages must supply functions that provide:

- **Initialization:** Most interpreters must be initialized before they can be used.
- **File Evaluation:** Every language must be able to evaluate code contained in a source file.

Unpublished

In addition, languages that support a top-level interpreter also need to provide:

- **Command completeness:** Since some commands can span multiple lines, the system must be able to determine when a command is complete so it can be evaluated (e.g., for Tcl, braces, parenthesis, brackets and quotes must match).
- **Prompt generation:** Each language must generate a prompt for the interpreter so that the user can identify which language is active and that input is needed.
- **String Evaluation:** Strings must be able to be evaluated as they are typed in to the interpreter.

To access our Pad++ library, each language must also specify two special functions.

- **Pad++ widget creation:** Whenever a new Pad++ widget is created, a function is called so that the scripting language knows about the new widget and can create a procedure for accessing the widget.
- **Pad++ command:** A special function is created in Tcl for each language. This allows each language to access the other languages. For instance, from Tcl, it is possible to get or set variables in Scheme.

Our implementation of access to multiple scripting languages in C++ is based on two classes, `Pad_Language` and `Pad_Script`. An instance of `Pad_Language` is created for each new scripting language. It contains pointers to six C++ callbacks that implement the functions described above. Then, every script, whether it is an event binding, or a command typed in the interpreter is instantiated as a `Pad_Script` with a pointer to the language the script is written in. The `Pad_Script` contains an `Eval()` method which evaluates the script in the appropriate language.

The only other special code deals with the top-level interpreter. An interpreter typically prints a prompt, receives input, and when the input forms a complete command, the command is evaluated. Finally, this cycle is repeated. Normally, these four steps are hard-coded for a specific language. With our approach, we simply call the appropriate call-back function for the current language.

The following sequence shows a very short usage of Pad++ moving back and forth between Tcl and Scheme:

```
unix> padwish
% puts "hello"
hello
% .pad settoplevel scheme
> (+ 2 2)
4
> (set! foo 42)
> (settoplevel 'tcl)
% .pad scheme get foo
42
% exit
unix>
```

DISCUSSION

One design tradeoff we made with this approach was the use of a string interface for parameter passing. As a result, all communication between each language and Pad++ goes through strings. This causes a minor speed penalty, but makes implementation *much* easier because internal types from each language only have to be converted to strings as a common intermediary. Therefore, each language can call the Pad++ library with a request in a standard string format (argc, argv) without having to know about each Pad++ library call and parameter type.

One difficulty we had is that not all scripting languages provide a well-defined and bug-free mechanism for converting internal types to strings. Since this is a very useful facility for the now common practice of embedding scripting languages in external programs, language designers would be well-advised to provide, test, and document string conversion facilities.

ACKNOWLEDGEMENTS

Thanks to the entire Pad++ development team for their patience and support as we figured out how to handle multiple languages reasonably.

This work is supported in part by ARPA contract #N66001-94-C-6039.

REFERENCES

[1] Benjamin B. Bederson, James D. Hollan, Ken Perlin, Jonathan Meyer, David Bacon, George Furnas. *A Zoomable Graphical Sketchpad For Exploring Alternate Interface Physics*, Journal of Visual Languages and Computing (in Press).

[2] Benjamin B. Bederson and James D. Hollan, *Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics,* Proceedings of ACM Symposium on User Interface Software and Technology (UIST'94), 17-26.

[3] Benjamin B. Bederson, Larry Stead, and James D. Hollan, *Pad++: Advances in Multiscale Interfaces*, Proceedings of ACM SIGCHI Conference (CHI'94), 315-316.

[4] John Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[5] ELK Scheme: http://www-rn.informatik.uni-bremen.de/software/elk/elk.html

[6] Perl: http://www.perl.com/perl

[7] STk: http://kaolin.unice.fr/html/STk.html

[8] Guile:http://www.cygnus.com/library/ctr/guile/guile.html

[9] Java: http://www.javasoft.com