Scripting For Java

Andrej Vckovski
Netcetera AG
Zypressenstrasse 71
8040 Zurich, Switzerland
andrej.vckovski@netcetera.ch

Michel Mathis
Netcetera AG
Zypressenstrasse 71
8040 Zurich, Switzerland
michel.mathis@netcetera.ch

ABSTRACT

Tcl has been initially developed as an embeddable command language to provide what we now call ”scripting” to complex applications. The ”scripting” or ”high level language” approach to provide control to applications from command lines, configurations files or ”macros” has been very successful and a major winning case for Tcl.

In the last six years, Java appeared as a programming language and runtime environment, and – due to many factors – gained a large popularity in the area of business computing. Therefore, the need to embed high-level languages for various reasons into Java-based applications was desired as much as it was for C/C++ and other low level language based applications. Java has sometimes been seen as a ”pariah” within the Tcl community, but still there are very useful projects and publications exploring various aspects regarding the relationship between Tcl and Java.

The Tcl/Java integration projects (tclBlend, Jacl) provide an embeddable Tcl interpreter written entirely in Java and a very powerful interaction between Java and the scripting level. As we have been and are using Tcl very much in our projects, Jacl would have been the logical choice for a scripting environment for Java. Nonetheless, we decided to do a comparison and evaluation project to compare Jacl with other popular scripting environments for Java such as Jython (having python as scripting language) and Rhino (JavaScript).

Here, we compare these environments based on a wide set of criteria such as popularity, library support, performance, ease-of-integration, memory footprints, licensing models and so on.

1. INTRODUCTION

Java has become a major programming language in the last six or seven years. There are many reasons why Java managed to establish itself that fast compared to other programming languages that needed many more years to find their way into the the “mainstream”.

Regardless of the corporate backing and marketing Java had and still has, the language itself and supporting infrastructure (libraries, development tools, deployment tools, runtime support and so on) has many useful features and properties, and it would be unfair to attribute its success only to the marketing of a few commercial organizations.

As a young company providing software development services to corporate clients in the area of Internet-oriented applications, Netcetera did very early start doing developments in Java. At the same time, we are using Tcl on a broad basis for our applications. Tcl has been designed to be embedded into larger applications, and that was what we were frequently doing. Using Tcl to provide

- powerful configuration facilities,
- command line interfaces to the end user,
- simple scripting of our applications abilities,
- gluing parts of the applications together,
- providing a customization layer for the user and more

proved to be a very powerful design “pattern”. For our Java-based applications, it was therefore natural to use the same design should it be required in a project.

There are a few comparisons of existing embeddable script interpreters for Java (e.g., [7, 9]), but we still decided that we have to do an evaluation of our own to come up with a decision on which one to use. As a company doing lots of Tcl development anyway, it seems to be very straightforward to use the Java-based Tcl-implementation (Jacl) right away without further investigations (assuming that there are no hard no-go issues).

However, Tcl itself has not had always a broad backing by some of our engineers, and so it would have been counted as a weird management decision if we would have decided to use Jacl, by-passing an evaluation of existing tools (actually it turns out that such emotional aspects do have a quite a large impact on these evaluations).

The first part of this paper discusses the methodology that we have planned to use for the assessment of the advantages and disadvantages of various approaches. The second part contains the actual results for three alternatives (Jacl, Jython and Rhino) we have evaluated in more depth. The various lessons we have learned are summarized in the last part together we the decision we have actually taken.

2. METHODOLOGY

2.1 Overview

In information technology and computer science it is a very common task to compare technologies that are meant to be suitable for a certain task. People compare hardware, operating systems, programming languages, frameworks, methodologies and much more. And it is very clear that the more complex or versatile a thing under evaluation is, the more difficult a comparison gets. For example, comparing the execution speed of processors is a fairly standardized task and there are well-defined benchmarks for that, maintained by independent benchmark standardization bodies. But even there, these standardized benchmarks often allow for much subjectivity when discussing the results. Comparing more abstract entities such as programming languages gets much more harder, and it is very typical to see long and heated debates on newsgroup once such a comparison is made (see also [10]).

We were very aware that our evaluation will not meet scientific precision nor will it be independent of our goals. Most of the criterias are driven and weighted by their fitness for our needs. It will become evident later on, that – even if we allow for a certain subjectivity ("we want to know which one is best for *our* goals") – any general statement is very difficult and ultimately, driven by personal taste and experience. Still, it is useful to compare the approaches, also to give possible indications for the various owners or maintainers where a potential development could be headed to.

The different criteria we investigated can be divided into three groups

- Hard criteria such as performance, memory footprint and so on
- Language features such as embedability, security aspects
- Soft criteria such as hipness, community support and acceptance, availability of third-party tools or literature

The following sections will define these criteria and our expectation, i.e., what value of that criteria is considered to be positive and what negative.

2.2 Hard criteria

2.2.1 Runtime performance

Whenever programming languages are compared, execution performance is frequently used as discriminator. There are very well-prepared performance benchmarks available (e.g., [1, 8]), having caused long discussions on various newsgroups. In the context of an embedded interpreter, execution speed is a somewhat delicate issue. That is, it very much depends on what the script interpreter is needed for. If you want simply a command line to be parsed with variable substitution and eventually dispatching into a function that is written in Java, execution speed will probably never matter. However, if you plan to run frequently large scripts of several hundred lines, it might be very well a critical issue. Generally, performance is not an issue if it is not perceived in any way by the user. If the parsing of a command line in a single-user application takes 10 milliseconds with system A and 60 milliseconds with system B, the difference will not be relevant. So if a system does reduce its complexity (e.g., by not having an embedded byte-code compiler) for the cost of lower execution speed, this trade-off might be very much in favor of that system.

Still, we performed a few academic performance tests in order to have relative values for the various environments. We measured:

- time to initialize interpreter subsystem
- time for first evaluation of a test script
- average time for a large number of follow-up executions

2.2.2 Code size

The size of the compiled Java code that needs to be included into the final application, i.e., the size of the interpreter does have insofar an impact as it does possibly blow-up the size of the host application[1]. So less is better.

2.2.3 Memory footprint

The interpreter should not add too much runtime memory usage to the host application. Java applications are often quite memory hungry and therefore, it matters if another add-on "wastes" too much primary storage. Here also, less is better.

2.2.4 Number of source code lines

The LOC (lines of code) of the interpreter implementation is a very rough measure for its complexity. As with most other criteria, it is easy to measure but hard to interpret. A large LOC value could mean:

- a lot of functionality (+)
- thorough implementation (+)

[1] We use the term *host application* for the application that does embed the interpreter

- higher risk for bugs (–)
- too much complexity (–)

2.3 Language features

2.3.1 Ease of embedding

We would like to have the ability to extend the host application with an interpreter without having to make too many compromises for the host application. I.e., the interpreter should not pose high requirements upon the host application. In earlier days for example, it was sometimes not too easy to embed classic Tcl into an application if the application had an own event loop (e.g., a GUI event dispatcher).

2.3.2 Sand-boxing the scripting language

Java has as mentioned before, very powerful introspection capabilities[2]. This offers a lot of possibilities when designing a language integration of, for example, a scripting language into Java. All script interpreters we have encountered did indeed use this capabilities and provide a lot of useful access to Java objects from the scripting side. However, you might want to control the amount of access that is allowed from the scripting side into your host application. Actually, in typical embedding cases (using the script language as a "command language" in the original sense of Tcl) you even want to be very explicit about what "commands" the host application offers. Also, it is frequently necessary to inhibit certain functionalities such as for example file operations.

2.3.3 Running in a sand box

In contrast to the previous criterion, it is sometimes also necessary, that the interpreter is functioning within a *Java* sand-box. I.e., within a runtime environment with lesser capabilities such as an unsigned applet in a Web browser. That is, it is required that the interpreter can limit its functionality so that it won't raise security violation exceptions. It would be a naive assumption, that if on the scripting side no "forbidden" things are used it will automatically please the Java sand-box. An interpreter might for example initialize certain subsystems (e.g., file access, getting the systems environment) even if they are not used later on. Here our test case is the ability of the interpreter to run in an *unsigned applet*.

2.3.4 Error messages and exceptions

When embedding an interpreter in the host application it will be also necessary to provide sensible feedback to the user for any evaluation errors such as syntax problems, undefined variables and so on. Since the parsing and evaluation is done by the embedded interpreter, it is also the interpreter that generates such exception messages. These messages need to be in a form that they can be intercepted and presented in a way defined by the host application. E.g., it might be necessary to provide translations into different languages. It is therefore not always desired to pass the interpreter's messages directly to the user. We would expect either a good categorization (by virtue of different Java exceptions, for example) or sensible, translatable error codes.

[2] Access to class properties such as attributes and their types, methods and their signatures, inheritance relationships and so on at runtime

2.3.5 Controlling input and output channels

As with the exceptions (2.3.4) and sand-boxing (2.3.2) mentioned above, the standard input and output channels of embedded interpreters must be controllable by the host application. E.g., the host application might want to display any output into a console window or a log file.

2.4 Soft criteria

2.4.1 Licensing model

The licensing model of the interpreter is of high importance when embedding the interpreter into an application. An embedded interpreter has the same role as any other "off-the-shelf" components that are used in an application and therefore, the same questions need to be asked beyond the technical capabilities of that component:

- Does the component and its licensing model increase the cost of the product?
- Do we have full control over the component, i.e., are we sure that the component does not have hidden features that impair the applications stability, security?
- Is there *fast* support available?
- Can we maintain the component even if its vendor goes out of business, gets bought by the competition or decides for whatever reason that the component is not anymore strategic?

Having mentioned these issues it becomes very evident that only an *Open Source* type of licensing model will provide sensible answers to these questions. Therefore, we did only investigate Open Source integration languages to start with.

2.4.2 Community maintenance

As a consequence of Open Source licensing we need to assess a product's maintenance from the community. Typical repositories for Open Source projects such as *SourceForge* usually provide numbers such as project activity figures that indicate whether a project is actively maintained. Also, the dates of the latest releases, mailing list activities are good indicators. It is important, however, not to misinterpret a project that did not release anything for a year as necessarily abandoned. There are, for example, many GNU projects that have not released anything for years because there is simply no need for a new version[3].

[3] Or consider TEX

2.4.3 *Ease of use*

Embedded scripting languages are most often intended to be used by the end users of the application, either as a command line interface or for automation and similar things. That is, the users do typically have no or very limited programming experience. Therefore, it is very natural to search for the integration language that is the easiest to use. Unfortunately, there are no simple criteria to assess how easy something is, and it is a clearly subjective assessment. For example, *Python*'s well known indentation feature for marking blocks has generated many heated debates between advocacy groups of various language camps. I would dare to claim that for non-programmers the meaningful indentation would be rather confusing than adding "better readability for humans", but I am sure there are as many people that are strongly convinced by the opposite.

2.4.4 *Availability of 3^{rd}-party documentation*

An embedded and integrated language in your application adds much power and elegant solution to customization, automation and other aspects. These add-ons will need documentation, support and training. When using a well-known language such as *Tcl* or *Python*, there will be a lot of literature available in book stores and on the Internet. There are also risks involved with external documentation. For example, *JavaScript* is today mostly used within Web browsers. Books on JavaScript therefore focus on that usage of JavaScript and often make no clear separation between language features and browser/document model features available *through* the language. As a matter of fact, it is even good that there is no clear separation, since the user (in our case of non-developers, power users) should not care whether something is a built-in function or provided by the host-application. For our user, *everything* is provided by the host application[4].

2.4.5 *Acceptance by developers*

The experience from many project shows that the acceptance by developers for any component or platform to can be a killer or success factor. If, for example, the database engine used is considered to be boring and legacy stuff, every problem in the project will eventually be due to the database. For an integration language, this acceptance issue is even more important. Most developers like to talk about languages, their shortcomings and great features, and most of them do also have strong opinions about what is good and what is not. Choosing a language which is considered to be bad, out, non-whatever-oriented or so can doom a project. However, it would be fatal if only such taste issues would dominate over more important criteria.

[4]A Tcl side note: Tcl as an language that was meant to be embedded *by design* does have many applications of itself where it is not called *Tcl* and where the users have no clue that they are actually using Tcl.

3. EVALUATION RESULTS

3.1 Candidates

The list of scripting languages for Java contains most of the currently widely-used scripting languages in general. The following list describes the candidates we choose for our evaluation and its motivation.

Jacl Jacl is part of the Java/Tcl integration set. It provides an almost full implementation of the Tcl 8.3. Jacl as a candidate was very natural: On the one hand, we have already been using the initial versions of Jacl in a project as early as 1998 and had some experiences with it. On the other hand, most of our development which is not Java is Tcl. So there is quite a large body of knowledge on Tcl available [11, 2].

Jython The python implementation in Java (was called JPython earlier) is an almost full implementation of the features of Python (with synchronized version numbers). Jython has not only gained interest for embedding purposes but also as a high-level replacement for Java, i.e., allowing to write programs in python that run on a JVM (Java Virtual Machine) and that use that large set of available Java packages [3].

Rhino Rhino is an implementation of JavaScript written in Java. The project was started at Netscape in 1997 as an embedded script interpreter for a Java-only browser. Meanwhile, Rhino is part of the Mozilla project [5].

Other languages There are many more scripting languages available for Java such as implementation of the *Ruby* and *Lua* languages, functional languages (*Lisp*, *Scheme*) and many more (a good overview is given in [12]). We have not included other languages in our evaluation for efficiency reasons.

The examples in figure 1,2 and 3 show the embedding of an interpreter. In the case of Rhino (figure 3) it is interesting to note that Rhino maintains a separate *scope* object. This allows an easy implementation of cases where a certain script (e.g., an event handler) can be executed in different scopes (e.g., for different windows or widgets, respectively).

3.2 Results

3.2.1 *Hard facts*

The test results for a few performance tests and the other quantifiable criteria mentioned above are summarized in table 1. These results need to be interpreted with care. The performance tests should only give a very rough impression and are far from being a well-calibrated and fair benchmark. Still, it shows that the performance of all three candidates are within a usable range. We did deliberately not include the detailed test scripts here to stress the qualitative nature of these comparisons. The values indicate normalized times, i.e., relative to the fastest of the corresponding category.

```
import tcl.lang.*;

public class JaclTest extends ScriptTest {

    static Interp interp;

    public void init() {
        interp = new Interp();
    }

    public Object execute(String s) throws Exception {
        interp.eval(s);
        return interp.getResult();
    }
}
```

Figure 1: Jacl (test driver)

```
import org.python.util.PythonInterpreter;
import org.python.core.*;

public class JythonTest extends ScriptTest {

    static PythonInterpreter interp;

    public void init() {
        interp = new PythonInterpreter();
    }

    public Object execute(String s) {
        interp.exec(s);
        // the script should leave the result
        // in the local variable 'result'
        return interp.get("result");
    }

}
```

Figure 2: Jython (test driver)

```
import org.mozilla.javascript.*;

public class RhinoTest extends ScriptTest {

    static Context cx = null;
    static Scriptable scope;

    public void init() {
        cx = Context.enter();
        scope = cx.initStandardObjects(null);
    }

    public Object execute(String s) throws Exception {
        Object result= cx.evaluateString(scope, s, "<cmd>", 1, null);
        return result;
    }

}
```

Figure 3: Rhino (test driver)

		Jacl	Jython	Rhino
version		1.2.6	2.1	1.5R3
released		April, 2001	December, 2001	January, 2002
performance				
initialization		*1.0*	2.4	1.2
single/infrequent execution		1.3	2.4	*1.0*
repeated execution		2.5	*1.0*	2.1
memory footprint	MB	3.9	7.1	4.5
code size	KB	784	1426	761
lines of code		52678	74863	56685

Table 1: Hard criteria

3.2.2 *Language-features*

Table 2 contains the summarized assessment of the language features compared. It turns out that all three candidates have fairly similar capabilities (where the numbers give a ranking).

3.2.3 *Soft criteria*

The soft criteria shown in table 3 do not show significant differences either. The numbers in the table are a subjective ranking between those three candidates.

All three interpreters are distributed under an open source license, with Jacl having the most liberal (BSD style; Rhino has Mozilla style and Jython a "Jython license", which is somewhat complicated because of inherited licenses).

The community maintenance in terms of project activity seems to be best with Rhino, but all projects have activities which where younger than 10 days at the time of this writing, so no one is "dead". When used for user level scripting of applications, we believe that Tcl provides the easiest access. Jython starts to be powerful when using the Java integration (accessing Java classes) and using object-oriented features which are not easy for non-programmers to start with. Also, Rhino is much more powerful as programming language than as "command line interpreter".

Developers on the other hand, like Jython very much because Python is considered hip. JavaScript has too much connotation of doing weird Browser-side scripting that is dependent on every minor web browser release and therefore, is not a very powerful name within the developer's community. For young programmers, Jacl seems to have inherited some of the dust Tcl is said to have.

3.3 So what?

The results shown above do not allow a quick and easy decision. As a matter of fact, we have just gotten the results of so many evaluations and comparisons done in the information technology community: Your choice depends very much on what you want to do, i.e., there is no general result possible because and the nifty details will later on turn to be either annoying or successful. We did actually not expect when starting the evaluation to have a clear winner in the end, but we have still been surprised that none of the candidates showed real show-stoppers that would have eliminated them.

So what to do? If the choice for a script interpreter cannot be derived from numbers as we engineers would like, other decision processes have to take place. While we'll be continuing this work with more in-depth analyses in the future, we decided for a current project to defer the decision by using IBM's *Bean Scripting Framework* (BSF, see also [6, 4]). BSF is an architecture for incorporating scripting into Java applications and applets that adds an abstraction layer on top of script interpreters. In theory, it is possible to change the script interpreter later on. To use such an abstraction framework does not only provide a generic solution, it is also actually a helper to overcome the decision problem if you don't dare to decide what subsystem to use. The same is, for example, true for database systems and the like, where a generic database layer does not necessarily mean that you want to replace the database in future, but it still generates less direct dependency and therefore, responsibility for the decision.

The usage of BSF allows us to continue our assessment on various script interpreters without having to wait on further results for our current projects.

4. CONCLUSIONS

4.1 Scripting for Java

Our experiences with a few larger Java applications with embedded scripting have shown that the design pattern of using high-level languages within lower level languages is very useful for Java as it was for C/C++ projects. The one-and-only style of marketing Java experienced in the last decade made many people believe that you have to decide between scripting and Java. Beyond the hype, we see that Java is a wonderful programming language and rich platform to develop complex systems, but it is not a silver bullet.

4.2 Evaluate?

When starting this evaluation we fell into the trap we experienced many times before. The set-up and question was easy: "What script interpreter should we use in the future for our Java applications?" During the subsequent evaluation process we realized again that the results we will get with a fair amount of time will not really help us in our decision. However, there were many side-benefits from our investigation, the most important maybe being our trust in all three script interpreters. No one really showed a big problem or annoyance. Evaluations of component for your applications in most cases lead to a negative selection (things you will definitely not choose) rather than a positive selection ("the winner is..."), and this is a useful result as well.

4.3 The case for Tcl

The Tcl–Java integration consisting of Jacl and tclBlend have been one of the first scripting implementations available in the Java arena. Both components provide very powerful scripting of Java applications, on-the-fly work with Java classes in tclBlend, using it as a unit test framework and so on. Yet, other approaches such as Jython or Rhino seem to have better resonance currently, not very different as it is in the "classic" Tcl arena. But still, it might be an opportunity that the connection to Java will help Tcl to gain again some more popularity.

5. REFERENCES

[1] D. Bagley. The great computer language shootout. `http://www.bagley.org/~doug/shootout`.

[2] Tcl java integration. `http://tcl.activestate.com/software/java`.

		Jacl	Jython	Rhino
Ease of embedding	rank	1	1	1
Sand-boxing the scripting language	rank	2	2	1
Running in a sand box	rank	2	2	2
Error messages and exceptions	rank	1	3	2
Controlling input and output channels	rank	2	1	3

Table 2: Language features

		Jacl	Jython	Rhino
Licensing model	rank	1	2	3
Community maintenance	rank	3	2	1
Ease of use	rank	1	3	2
Availability of third-party documentation	rank	1	1	1
Acceptance by developers	rank	3	1	2

Table 3: Soft criteria

[3] Jython. `http://www.jython.org`.

[4] Bean scripting framework. `http://oss.software.ibm.com/developerworks/projects/bsf`.

[5] Rhino: Javascript for java. `http://www.mozilla.org/rhino`.

[6] M. Johnson. Script javabeans with the bean scripting framework. `http://www.javaworld.com/javaworld/jw-03-2000/jw-03-beans.html`, March 2000.

[7] D. Kearns. Java scripting language: Which is right for you? `http://www.javaworld.com/jw-04-2002/jw-0405-scripts.html`, April 2002.

[8] B. W. Kernighan and C. J. V. Wyk. Timing trials, or, the trials of timing: Experiments with scripting and user-interface languages. `http://cm.bell-labs.com/cm/cs/who/bwk/interps/pap.html`.

[9] R. Laddad. Scripting power saves the day for your java apps. `http://www.javaworld.com/jw-10-1999/jw-10-script.html`, October 1999.

[10] C. Laird. Cameron laird's personal notes on language comparisons. `http://starbase.neosoft.com/~claird/comp.lang.misc/language_comparisons%.html`.

[11] I. K. Lam and B. Smith. Jacl: A tcl implementation in java. In *Proceedings of the 5th Annual Tcl/Tk Workshop*. USENIX, July 1997.

[12] R. Tolksdorf. Programming languages for the java virtual machine. `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

6. ACKNOWLEDGMENTS

Many other persons at Netcetera supported various aspects of this work, especially Joachim Hagger,Simon Hefti and Jason Brazile.