JTcl and Swank: What's new with Tcl and Tk on the JVM

[1]Bruce A. Johnson, [2]Tom Poindexter, & [3]Dan Bodoh

[1] bruce@onemoonscientific.com, One Moon Scientific, Inc, Westfield, NJ and University of Maryland, Baltimore County
[2] tpoindex@gmail.com
[3] dan.bodoh@gmail.com

Abstract

JTcl is an implementation of the Tool Command Language (Tcl) written in Java and is derived from the Jacl project. The current release (2.0) of JTcl implements a large extent of Tcl 8.4 syntax and commands, limited only by the API restrictions of the Java Virtual Machine. Swank is an implementation of the TK GUI toolkit implemented using the Java Swing GUI API. Most Tk 8.4 widgets and commands have been implemented as well as additional ones based on Swing widgets. This paper describes the current state of these projects and gives examples of their use.

Introduction

The Java Virtual Machine [Lind99] has become a platform on which a variety of computer programming languages can be executed. While originally written to execute Java programs that had been compiled into Java byte codes, it is now used to run languages such as Clojure, Groovy, Jacl, JRuby, Jython, Rhino and Scala [WikiJVM]. Some of these, like Scala, appeared originally as a language on the JVM, and others, like Jacl, are JVM implementations of existing languages. Jacl, which is an implementation of Tcl [Ost10], was one of the first non-Java languages on the JVM and appeared shortly after the initial development of Java [Lam97].

While there is an abundance of alternative programming languages on the JVM, there are relatively few implementations of graphical user interface toolkits besides the AWT and Swing toolkits that come standard with most Java distributions. The primary alternatives have been Swank, SWT (the Standard Widget Toolkit), and quite recently JavaFX. While SWT is largely implemented on top of native platform widgets and JavaFX is implemented with its own windowing toolkit, Swank is a layer on top of the Swing widgets that provides an interface to the programmer that is analogous to that of the Tk toolkit [Ost10].

In this paper we'll discuss recent developments in the Tcl and Tk on the JVM, focussing on the language implementation JTcl, and the Tk-style graphical user interface toolkit Swank.

JTcl

JTcl is a fork of Jacl, an implementation of Tcl written in Java [Lam97]. Jacl was written during the period of Tcl/C 8.0 development and contains Java equivalents of many internal data structures, most importantly the notion of Tcl objects to hold binary representation of numeric data types, efficient list and array structures, and implementation of most of Tcl 8.0's commands. Jacl does not implement the Tcl byte code compiler and runtime [Lew96], nor the Tcl fileevent command and supporting event system that allows for event driven I/O. After initial development by the original authors, Jacl development was performed by individuals rather than as an official port of the Tcl Core Team. During this time Jacl development slowed to mostly bug fixes but did result in a few major improvements, a port of the Incr Tcl object system [DeJ05] and the Tcl-to-Java compiler (TJC) [DeJ06]. Development of core Tcl commands and features did not keep synchronized with mainline Tcl development. Despite Jacl's slow progress, it had proved useful in a number of commercial products, open source projects and proprietary internal projects. Jacl was used by IBM in its WebSphere Application Server and One Moon Scientific's NMRView products [John04], as well as the open source Swank and Æjaks projects [Poin07].

Jacl Modernization

Jacl modernization was selected as one of the Tcl Core Team projects during the Google Summer of Code 2009 [Szul09]. The goal of the project was to bring Jacl's language features to the level of Tcl/C 8.4. Tcl 8.4 was chosen as a target level for several reasons. First, it represented a stable base of Tcl compliance that could be achieved by implementing new commands or augmenting existing ones. Second, the project was limited to one student for one summer, so the work product of the GSOC project was limited. Third, current Tcl/C version 8.5 contained many structural changes, such as the expand syntax which would require a considerable amount of interpreter changes. The target of the GSOC project was derived by comparing the current set of Jacl command definitions with the Tcl/C 8.4 definitions. Many of the command implementations required relatively little additional code to support a particular command option, larger code rework was required to implement commands such as [regex] and [regsub]. These commands relied on moving from a custom underlying regular expression library to use the Java **java.util.regexp** package.

While the GSOC 2009 Jacl Modernization project yielded many improvements, it did not reach full Tcl 8.4 compliance. The GSOC project relied on command descriptions based on the Tcl 8.4 manual pages, so while many command options were added or improved, strict compliance to Tcl/C test cases was not tested. The Jacl implementation of [regex] and [regsub] improved significantly to match Tcl 8.4, but many edge cases were not addressed. Addition of a event system and [fileevent] also proved to be too ambitious, requiring more time that was available.

The JTcl project was formed to continue development of Jacl and complete the work of the GSOC Jacl Modernization project. The project founders decided that a fork was the best way to achieve its goals. Jacl was a part of the TclJava project, which produced the the Jacl interpreter as well as TclBlend, a Tcl extension that enables use of Java classes and objects from the the Tcl/C runtime. Much of the TclJava packaging and build system was designed to support the use of the java package in both Jacl and TclBlend environments. JTcl project members had no interest in the TclBlend extension and instead would focus entirely on the Java implementation of Tcl.

In addition to furthering Tcl 8.4 compliance, a number of other improvements were desired. First, Java code development is greatly enhanced by the use of Java-centric Integrated Development Environments (IDEs) such as Eclipse, Netbeans and IntelliJ, so the structure of the JTcl source code should be arranged to support easy use by Java IDEs.. Second, the build system in Jacl using *make* would be replaced with a Java-centric build system. While *make* could be used to compile and package JTcl, Java oriented build systems *ant* and *maven* are better supported by Java IDEs. Third, packaging the JTcl system would be in a single jar file for simple installation, as opposed to the Jacl system packaging in five separate jar files. Lastly, extraneous source code such as the TclBlend extension would be removed entirely.

Tcl Compliance and Test Suite

The Tcl language for any particular version is described in man pages and other documentation, but the definitive source of Tcl compliance is represented by the Tcl test suite. The test suite is usually developed in conjunction with a particular version of Tcl to ensure that the interpreter's result for any give operation matches expected results.

The Jacl project contained a test suite that matched Tcl 8.0 compliance and was enhanced as changes were made to the source code. For JTcl, the Tcl 8.4 test suite was imported and used to measure Tcl compliance. JTcl integrates the Tcl test suite through the JUnit test facility. JUnit is a Java oriented test environment, roughly equivalent to the tcltest Tcl package. When running JUnit in a Java environment, the normal usage is to run a test method that invokes methods on an object under test, and asserts that actual results are equal to expected results. Since JUnit is widely supported by Java IDEs, the Tcl test suite in JTcl is invoked through JUnit classes. This allows testing of JTcl source code directly from the IDE, without requiring a compile/test cycle.

The Tcl test suite contains generic tests that should run the same on any execution platform as well as many tests that are specific to the platform. For example, a particular test may only run on a Windows platform, while an equivalent test may only run on a Unix environment. A Java JVM presents a single virtual machine that (mostly) eliminates machine and platform differences. As a result, only the tests that are labeled as generic are tested in JTcl.

Even with running only the generic Tcl tests in JTcl, many differences in test results were observed and many of which were false negative results. Erroneous test results generally fell into the following categories:

1. **Differences in error messages** – when a test would check for specific error messages, differences between JTcl and Tcl/C would often arise as error callback messages may contain slightly different text. Most of these differences are a result of Tcl/C's byte code compiler, which returns errors stating “...while compiling...” vs. the pure interpreter's error messages “...invoked from within....”.
2. **Ordering of results** - many Tcl commands return unordered results, e.g. [info commands]. Due to JTcl's use of native Java libraries for hash maps instead of Tcl's C coded ones, key lists were returned with different orderings.
3. **Unsupported functions of the JVM** – the Java JVM does not support many low level system functions, so Tcl commands such as [file stat] are limited to the operations that can be performed.
4. **Regexp differences** – JTcl makes use of the Java library **java.util.regexp** package for regular expression handling, whereas Tcl uses the Spencer ARE library coded in C. While most common Tcl ARE regular expressions are accepted in JTcl via direct use of **java.util.regexp** or through emulation, some Tcl ARE expressions such as the Basic-RE meta-character ('b') are not supported.

To work around these differences, the JTcl JUnit base class is designed to run a Tcl test suite test file with a list of expected failure cases. Each failure case returned by the test suite is examined to note the type of the failure, and when the difference could be categorized as one of the above cases, that case was added to the expected failure list. The result of the expected failure lists allow the entire test suite to be run, with a better indication of positive or negative results. Numerous Tcl command implementation classes were modified to pass the Tcl test suite.

Code Modernization / IDE support

While the main focus of the JTcl project is to continue the effort of making JTcl conform to the Tcl language 8.4 test suite, and number of other efforts were done to modernize the code. “Modernize” is somewhat a subjective term. The JTcl project's definition of modernization includes reforming the code as if the JTcl code was being developed new by skilled Java programmers using accepted Java development best practices and tools.

The existing Jacl Java code was originally developed to closely mimic the Tcl/C version. This was likely done for ease of the initial port to Java. JTcl has the following changes to the source code, besides those made for test suite compliance:

1. **Source packages** – Java code can be organized into distinct packages (i.e., namespaces). This promotes grouping similar source code classes by function. In JTcl, the package tcl.lang.* is

used for core interpreter classes, tcl.lang.cmd.* for commands, tcl.lang.channel.* for I/O classes, etc. Standard JTcl packages java, itcl, and tjc were moved to tcl.pkg.* packages. Included Tcl library code (e.g., *.tcl files) was moved from Java source code directories to resource directories.

2. **Code formatting** – much Jacl's source code had specific hand-formatted conventions, such as ASCII form-feed characters (^L) to separate methods, comments within method arguments, debug-only code fragments. JTcl code is reformatted using automated tools for consistency, and debug specific code is removed in favor of using the IDE's debug and breakpoint facilities.
3. **Block comments converted to Javadoc comments** – Jacl code contains many block comments that precede methods, but these were not in the format to support the Javadoc tools for creating automated source code documentation. Where practical, source code block comments in JTcl are Javadoc formatted.
4. **IDE/build tool friendly directory layout** – the project directory layout was changed to easily support Java IDEs and build tools. src/main/java contains the Java source code, src/main/resources contain Tcl library code, src/test/java contains Java JUnit code, src/test/resources contain test Tcl code (i.e., the Tcl test suite.) Additional directories contain the project website source code, maven assembly descriptors, runtime startup scripts, etc.

Packaging / Tcllib
Recent Jacl distributions have included the Incr Tcl and Tcl-to-Java compiler packages. Jacl's packaging favored separate jar files for the Jacl core interpreter and each extension. JTcl instead packages all core and extension components into a single jar file. Jacl also includes Tcllib as part of its packaging. Tcllib is a large collection of Tcl coded libraries.. Some modules of Tcllib that only support Tcl versions 8.5 and 8.6 are excluded in the JTcl distribution.

Packaging all core and library components of JTcl into a single jar file allow the interpreter to be started as simply as java -jar jtcl.jar, though a more common usage still utilizes helper scripts. The JTcl startup scripts jtcl (for Linux/Unix/MacOSX/Cygwin/Msys environments) and jtcl.bat (Windows) allow for additional jar files to be included via the normal CLASSPATH environment variable, as well as runtime Java JVM parameters to be easily modified.

The JTcl website is included in the project and is built using the maven build system. The JTcl source Javadoc files are also built during website generation.

RegExp Improvements
The regular expression engine class, tcl.lang.Regex, is new in JTcl and used by [regsub], [regexp] and [lsearch -regexp]. This class brings the full power of TCL 8.4 Advanced Regular Expressions (ARE) to JTcl, with a few caveats. The older Basic Regular Expressions (BREs) and Extended Regular Expressions (EREs) are not supported, although EREs that are identical to AREs and not explicitly re-

quested with the 'e' embedded option are supported.

A primary implementation goal was to make use of **java.util.regex.Pattern** and **java.util.regex.Matcher** [Oracle2004], rather than writing a custom engine based on the C library used in Tcl 8.4. A **tcl.lang.Regex** instance combines the steps of compiling a regular expression and matching it on an input string, and contains all the functionality required to implement [regsub] and [regexp].

The **tcl.lang.Regex.compile()** method is responsible for converting a Tcl regular expression to a Java **Pattern** instance. This method parses the Tcl regular expression, building a Java regular expression in a **Stringbuffer**, and compiles the Java regular expression into a **Pattern** instance.

Many aspects of the conversion of Tcl regular expression syntax to Java syntax are merely direct translations. For example, a static Hashmap is used to translate Tcl's character classes and escape sequences to Java's equivalent, such as [:alnum:], to \pAlnum and [:ESC:] to \\033.

More complex translations are required for those elements that are similar in the two regular expression languages, but differ in minor details or at boundary conditions. For example, an empty Tcl regex matches before every character in the string, and after the last character. Java's empty regex is similar, but does not match after the last character. So a Tcl empty regex is translated to ^|(?!$) for Java. Many similar complex translations are needed for the embedded options, which are similar to but not exactly like the **java.util.regex.Pattern** match flags.

Tcl does contain some regex features that are not available in Java. These are emulated with more complex Java expressions. For example, the Tcl \M (match at the end of a word) has no direct Java equivalent, so it is translated to (?=\W|$)(?<=\w) (look ahead for a word character and behind for non-word character).

The decision to use **java.util.regex.Pattern** led to one incompatibility in JTcl regular expressions. Tcl always attempts to match the longest string starting from the outermost levels to the inner levels of parentheses. With alternation (A|B) Tcl chooses the longest match of all the branches. Java evaluates the regular expression from left to right, and returns the first successful match, even if it's not the longest. This incompatibility will not affect most common uses of [regexp] and [regsub].

Pattern syntax error information returned by Tcl is replicated by translating the message from the **java.util.regex. PatternSyntaxException** thrown by the Java **Pattern.compile().** However, Java is more forgiving about poor regular expression syntax, and therefore some expressions that would generate an error in Tcl may be interpreted as literal characters in JTcl.

Code refactoring was done to collapse the [regexp] and [regsub] common code into **tcl.lang.Regex**. The matched input substring state information used by [regexp

-all] and [regsub -all] was delegated to **java.util.regex.Matcher**, simplifying the code.

An apparent Tcl 8.4 bug was replicated in the JTcl code: a difference between regexp and regsub. The command [regexp -all -inline {a*} {a}] returns one match, {a}. The similar command [regsub -all {a*} {a} {Z}] returns {ZZ}, one Z for the match of {a} with {a}, and a second Z for a zero-length match after the 'a'. The **java.util.regex.Matcher** match groups are used for code simplicity, with a special case in the [regexp] implementation for this inconsistency.

Process Pipelines for [exec] and [open]

Process pipelines for [exec] and [open "| command"] and the Tcl 8.4 [exec] input and output redirection were added to JTcl, using pure Java. The **tcl.lang.Pipeline** class parses an [exec]- or [open]-style pipeline string and builds a chain of **tcl.lang.process.TclProcess** instances for the chain of operating system commands in the pipeline. Each **TclProcess** instance is made aware of its neighbor **TclProcess** (or its redirected input and output) with a **tcl.lang.process.Redirect** instance. The **Pipeline** instance can manage any of the [exec] redirectors The [open "| command"] command uses a channel view of **Pipeline**, **tcl.lang.channel.PipelineChannel**.

The **tcl.lang.process.TclProcess** class is abstract with currently one concrete subclass: **tcl.lang.process.JavaProcess**. **JavaProcess** is a pure Java implementation using **java.lang.Process** and **java.lang.ProcessBuilder**. This code organization allows for future development of platform-specific **TclProcess** subclasses that use native code, or a Java 7 subclass that makes use of the new redirection capabilities of **ProcessBuilder**.

The **tcl.lang.process.TclProcess** subclass is responsible for handling its own input and output redirection. **JavaProcess** is limited by the capabilities of the Java 1.5 and 1.6 API, which does not expose the operating system's pipe and file descriptor inheritance mechanisms. All pipelines and redirection must use **Process.getInputStream()**, **Process.getOutputStream()**, and **Process.getErrorStream().** To create a pipe, a new thread is created with an instance of **tcl.lang.process.TclProcess.Coupler** which reads the upstream **JavaProcess**'s output stream and writes to the downstream **JavaProcess**'s input stream.

These limitations in the Java API create some incompatibilities between a Tcl pipeline and a JTcl pipeline. A pipeline launched in the background by JTcl cannot outlive the JTcl process itself because JTcl, rather than the operating system, is managing the pipe.

The Java Process API use of the **InputStream** class for standard input and the lack of file descriptor inheritance in the API creates problems for JTcl's tclsh emulation, **tcl.lang.Shell,** when using [exec]. With Java's **InputStream**, the only way to detect an end-of-file condition is to do an **InputStream.read().** But doing the read will take at least one byte from the standard input. So the **JavaProcess** instance for an exec'd process can take an extra byte from standard input that it may not need, stealing that byte from the JTcl shell itself. A simple example is shown below.

Contents of the file testStdin.txt:

```
exec head -1
this line should go to head and to
stdout
puts {this line should be inter-
preted by the JTcl shell}
exit
```

This file is sent to the JTcl shell via standard input:

```
$java tcl.lang.Shell <
testStdin.txt
```

Two possible cases occur – the first is the expected output:

```
this line should go to head and to stdout
this line should be interpreted by the JTcl
shell
```

The second case is when the JavaProcess instance for 'head' steals an extra byte

```
this line should go to head and to stdout
couldn't execute "uts": no such file or di-
rectory
```

These incompatibilities are relatively minor, and could be fixed with the Java 7 capabilities of **ProcessBuilder** which support true file descriptor inheritance at the operating system level.

The [pid fileid] command is supported on Posix systems on at least some JVMs by looking for a field named "pid" in the java.lang.Process instance with a value the same as that returned by **Process.getClass().getDeclaredField("pid")**. If this fails, -1 is returned as the process id.

File Events and the New Channel Subsystem

Significant improvements were made to the channel subsystem for JTcl to support non-blocking I/O, Unicode, and to fix failing tests in the Tcl 8.4 test suite. Both [fcopy] and [fileevent] are supported,

The [fcopy] command simply copies from one channel to another within a separate Java thread, and uses the existing JTcl event queue to execute the callback script when [fcopy] completes. If possible, a byte copy is made to avoid the Unicode encoding and decoding step, and an efficient buffering is enabled.

The [fileevent] command depends on the new non-blocking I/O implemented in the channel subsystem. The fileevent itself is described with two objects, **tcl.lang.channel.FileEventScript** and **tcl.lang.channel.FileEvent**. The instance of **FileEventScript** exists for the lifetime of a fileevent, and schedules new instances of **FileEvent** on the JTcl event queue. Each **FileEvent** instance, when it comes off the queue, tests for readability or writability of the channel and executes the fileevent script as necessary.

As originally coded in Jacl, the **tcl.lang.channel.Channel** abstract class is the root object for all types of channels. Much of the channel code was re-written in a more Java-like fashion, replacing the literal C-to-Java translation. Subclasses of **Channel** are shown in Table 1.

Table 1	
Java Class	**Description**
SeekableChannel	Abstract class that adds seek() and tell()
FileChannel	Extends **SeekableChannel** to implement file I/O
ResourceChannel	Implements reading of a Java resource using a “resource:” prefix on the file name
ReadInputStreamChannel	Bridges a Tcl channel to a Java **InputStream**
AbstractSocketChannel	Abstract class that has common code for socket channels
ServerSocketChannel	Implements Tcl server sockets
SocketChannel	Implements Tcl sockets
TclByteArrayChannel	Used internally to bridge Tcl channels to Tcl byte arrays

In order to support the JTcl enhancements and fixes, Jacl's **TclInputStream** and **TclOutputStream** classes were replaced with a chain of subclasses of **java.io.InputStream**, **java.io.Reader**, **java.io.OutputStream**, and **java.io.Writer**.

The input side of a Channel uses the following chain of **InputStreams** and **Readers**:

Channel.getInputStream() presents an **InputStream** view of the data on the channel. For example, a **FileChannel** uses **java.io.FileInputStream**.

EofInputFilter reads bytes from **Channel.getInputStream()** and adds the end-of-file byte configured by the channel.

InputBuffer reads bytes from the EofInputFilter, provides a resizable read buffer and implements non-blocking reads. It performs non-blocking reads by performing **EofInputFilter.read()** in a separate thread. All byte read operations on the channel are taken from this **InputStream**.

MarkableInputStream reads bytes from the **InputBuffer** and allows for look-ahead in the stream.

UnicodeDecoder reads bytes from the **MarkableInputStream** and converts to Unicode using the encoding configured by the channel.
EolInputFilter reads characters from **UnicodeDecoder** and performs the configured end-of-line translation on the

channel. All character read operations on the channel are taken from this **Reader**.

The output side of a **Channel** uses the following chain of **OutputStreams** and **Writers**:

EolOutputFilter is written to by the channel when it performs character writes. It performs the configured end-of-line translation on the channel.

UnicodeEncoder is written to by **EolOutputFilter**, and translates Unicode characters to bytes according to the encoding configured on the channel.

OutputBuffer is written to by **UnicodeEncoder** as well as by the channel when it performs byte writes. It provides a resizable buffer, but unlike **InputBuffer**, does not handle non-blocking writes.

EofOutputFilter is written to by **OutputBuffer** and adds the end-of-file character that the channel is configured to use.

NonBlockingOutputStream is written to by **EofOutputFilter**, and performs its **OutputStream.write()** and **OutputStream.flush()** in a separate thread for non-blocking writes.

Channel.getOutputStream() is written to by NonBlockingOutputStream, and provides an OutputStream view of the channel data.

Testing Sockets and File Events

A hallmark of Tcl is its event system that allows writing of servers with a minimal amount of code. An example of this is the DustMote script [Kapl02] that implements a web server in merely 41 lines of code. We found that DustMote running under JTcl could readily serve a web site (the document root was set to the content of the www.onemoonscientific.com site) indicating that the fileevent and server socket code functions as expected.

As a further test, multiple simultaneous instances of JTcl were set up calling a script using the [http::geturl] command to pull a file from the DustMote server. As described above the fcopy command initiates a separate Java thread to do the file copy to the clients socket and we indeed observed that the Thread usage by DustMote increased proportionally to the number of clients accessing it.

Swank

The success of Tcl as a programming language comes not only from the intrinsic value of Tcl, but its companion Graphical User Interface Toolkit, Tk. Tk has become so successful that it is used not only as the GUI toolkit for Tcl, but also with other languages such as Python. Without a Java implementation of Tk, JTcl would not be able to fill many of the programming niches accessible to Tcl. Tk widgets are, however, programmed with low level calls to each platforms native graphics system and replicating this in Java would be a large task.

Developing Swank

Two key factors allowed for the feasibility of developing Swank ("Tk in Java") in a reasonable period of time. Swing, the primary Java user interface toolkit, provides a rich variety of widgets with similar functionality to Tk widgets. For example, the Tk toplevel widget is similar to the Swing

Table 2			
Swing	Tk	Swing	Tk
JButton	button	JRadioButtonMenuItem	radiobutton (on menus)
JCheckBox	checkbutton	JScrollBar	scrollbar
JFrame	toplevel	JSlider	scale
JLabel	label	JSpinner	spinbox
JList	listbox	JTextArea	message
JMenu	menu	JTextField	entry
JMenuBar	menubar	JTextPane	text
JPanel	frame	JFrame (composite)	labelframe
JRadioButton	radiobutton	JPanel (customized)	canvas

JFrame widget, the button to JButton, the menu to JMenu, etc. Using the Swing widgets meant that the behavior of Swank would not be as similar to Tk as it would if the Swank widgets were developed with lower level Java graphic operations. On the other hand, adopting Swing meant that a great deal of coding work could be skipped. Furthermore,using the Swing widgets provides a richer set of behaviors than the original Tk widgets.

The second key factor was the introspection capabilities of the JTcl language. Much of the code that forms the basis of Swank is generated by JTcl scripts that determine the fields and methods of each Swing component and then automatically produce Java code that provides a Tk-like interface to the components. This generates a large number of configuration options for each widget. Some of these map

Table 3			
Swing	Tk	Swing	Tk
JDesktopPane	jdesktoppane	JProgressBar	jprogressbar
JComboBox	jcombobox	JScrollPane	jscrollpane
JDialog	jdialog	JSplitPane	panedwindow
JEditorPane	html	JTabbedPane	jtabbedpane
JInternalFrame	JInternalframe	JTable	jtable
JOptionPane	joptionpane	JToolBar	jtoolbar
JPasswordField	jpasswordfield	JTree	jtree
JPopupMenu	jpopupmenu	JWindow	jwindow

coincidentally to the names and functions of Tk configuration options. In other cases, JTcl code is used to specifically generate Java code for Tk options. In some of these cases it is only necessary to generate code that parses the appropriate Tk option and maps it to an existing Java Swing method. In other cases specific Java code is written to enable the correct action in response to the specified option. This Java code is inserted in the generated Java file.

In earlier versions of Swank we made available nearly all configuration options of the Swing widgets as Tk-style configuration options. Starting with version 3.0 the code generator has been changed to limit the options to a predefined list that leaves out many of the more obscure Swing configuration options. This leads to a simpler toolkit that presents options more consistent with that of the Tk toolkit.

Swank Widgets
Swing widgets and the Tk style commands used with Swank to create them are listed in Table 2. These are the widgets that have a particularly close correspondence between the Tk widget and the Tk-style widget as implemented in Swank.

Some Swing widgets don't have a direct correspondence to existing Tk widgets, but were deemed useful enough that they should have a Tk style command in Swank. These are listed in Table 3. Some of them do have analogous Tk commands that are available in extensions like the table and combobox widgets. Others, like the panedwindow, exist in Tk, but the Swank implementation has significant differences.

The behavior of most of the widgets in these tables is largely a product of that of the underlying Swing widget. The two most complex Tk widgets, text and canvas, required substantial Java code to reproduce the behavior of the Tk widgets. The canvas widget, in particular, is almost entirely implemented by Swank specific Java code. This widget is based on the Swing **JPanel**, which essentially provides an empty screen area on which to draw by overriding its **paintComponent** method.

Swank Canvas Widget
The Swank canvas widget provides most all of the features of the Tk canvas, plus some additional capabilities. Colors are one area where the Swank canvas is distinguished from that of Tk. In Swank, objects like rectangles and ovals can have gradient or texture fills, and the colors for all Swank canvas items can be transparent.

Configuration Options
Additional configuration options are available for Swank canvas items. For example, while Tk lines can have arrows at one or both ends of the line, lines on the Swank canvas allow for different styles (arrow, square, circle, diamond or nothing) at each end. All Swank canvas items also support a -rotate configuration item. A common style when generating diagrams is the placing of a text label on a shape. To facilitate this, rectangles and ovals on the Swank canvas can be configured with a text option (and corresponding font and text color options).

Additional Canvas Items
Several additional canvas item types are present in Swank. In addition to normal text items, the Swank canvas adds htext items. These support many HTML tags and some CSS styles (as implemented by Java Swing HTML endowed text widgets). For example, an htext item could have an H2 header, superscripts, bold and italic text or be laid out as a table using HTML table tags.

Connection items are unique in that their coordinates are specified in terms of a fraction of the bounds of two other items on the canvas. In this way it is easy to produce diagrams where dragging one item around maintains a displayed connector to a second item without needing to write Tcl level code to reposition the connector. Annotation items combine a line with an arrow at one end and a text string at the other.

Affine Transforms
All Swank canvas items can have an Affine transform associated with them. The standard Swank canvas includes fractional transforms that allow canvas drawing in fractional positions of the canvas, allowing, for example, a rectangle to fill the top half of a canvas, no matter how the canvas is resized. This capability is extensively used in the NMR analysis program dataChord where custom canvas items add transforms to the canvas that allow items to be drawn relative to the first items position. In this way labels and annotations positioned near features of the NMR spectrum remain positioned relative to the NMR feature, no matter how the whole spectrum is zoomed or panned. Additionally, the whole Swank canvas has an Affine Transform associated with it, the scale of which is changed with the canvas "zoom" subcommand. This allows one to zoom the view of the entire canvas in or out.

Handle Selection
A standard feature of many programs for creating diagrams or illustrations is the ability to select, move and resize items on the drawing canvas using various mouse actions. This can be implemented in a Tk program by drawing selection indicators and handles with explicit canvas items, but it seemed such a common paradigm that we added low level support to the Swank canvas for these actions.
All items can be selected using an "hselect" subcommand. Any items that are selected are displayed with selection handles. When the mouse enters a handle the cursor is changed to an "appropriate" resize cursor. The handles are not implemented as separate canvas items, but are fundamentally displayed by the underlying Java code at appropriate positions on the bounds of the item. Moving and resizing selected items is the responsibility of "user code" and is not part of Swank, but is easily implemented.

Scene Graph
Advanced graphics applications often arrange the display items as a collection of nodes in a graph structure known as a Scene Graph. Rendering of the items is then done by traversing the scene graph and rendering each viewable item. Whether or not items are rendered in front

of or behind other items depends on their relative position in the scene graph. A scene graph is being developed for the Swank canvas. As currently implemented the Swank canvas scene graph is implemented by adding a new "-node" configuration option for each item on the canvas and adding a new node item type. Each traditional item on a canvas (arc, rectangle, line etc.) exists as leaf node on the graph and can not have other items attached. Only the new node item can have descendants, which may be traditional display items, or additional nodes.

If nodes are not specified the canvas acts as the traditional Tk canvas, effectively being a scene graph with one root node and zero or more visual items that are rendered in order of their attachment to the root node. The scene graph is rendered in a depth-first (post-ordered) fashion with children at each node rendered from left to right (first to last added). The bounding box (returned with the "canvas bbox" command) is the union of the bounds of all the items below that node. Node items are also rendered if they have a non empty fill or outline parameter. They are rendered as rectangles whose size is the same as the bounding box described above. Note that if the fill parameter is set, and is not transparent, all items below that node will be obscured as the node is drawn after the items below it on the graph.

The raise and lower canvas subcommands have a modified behavior with respect to scene graphs that have more than one node. A raise command issued without a "aboveThis" argument will move the specified items to be the last items of the node to which they are attached. If an "aboveThis" argument is specified, the aboveThis item must be attached to the same node as any items to be moved. Thus raise (and the comparable holds for lower) will only change the display order of items relative to other items attached to the same node (but, note that node items themselves can be raised or lowered).

Charts

Charts are implemented using JFreeChart [Gilb11]. In the Swank implementation they are essentially just another item that can be placed on the canvas. The chart shown in Figure 1, for example, is not implemented by using a multiple individual canvas items, as would be done in Tk, but is instead a single chart item that can readily be resized and repositioned on the canvas.

The canvas charts, illustrate a significant advantage of working in the JVM environment. Working with the C implementation of Tcl/Tk one would need to find a charting library that works on all the major environments (Mac, Windows, Linux, etc.) and then ensure that it compiles, links and runs on these. Integrating such a library might require significant knowledge of the build environment of each operating system. Using such a library also requires an ongoing commitment to update the library and build environment for new operating system releases. With the JVM approach,however, one needs only ensure that the libraries jar files are available on the build and run classpaths and one has a high level of confidence that the application will run on any platform implementing a compatible version of the JVM.

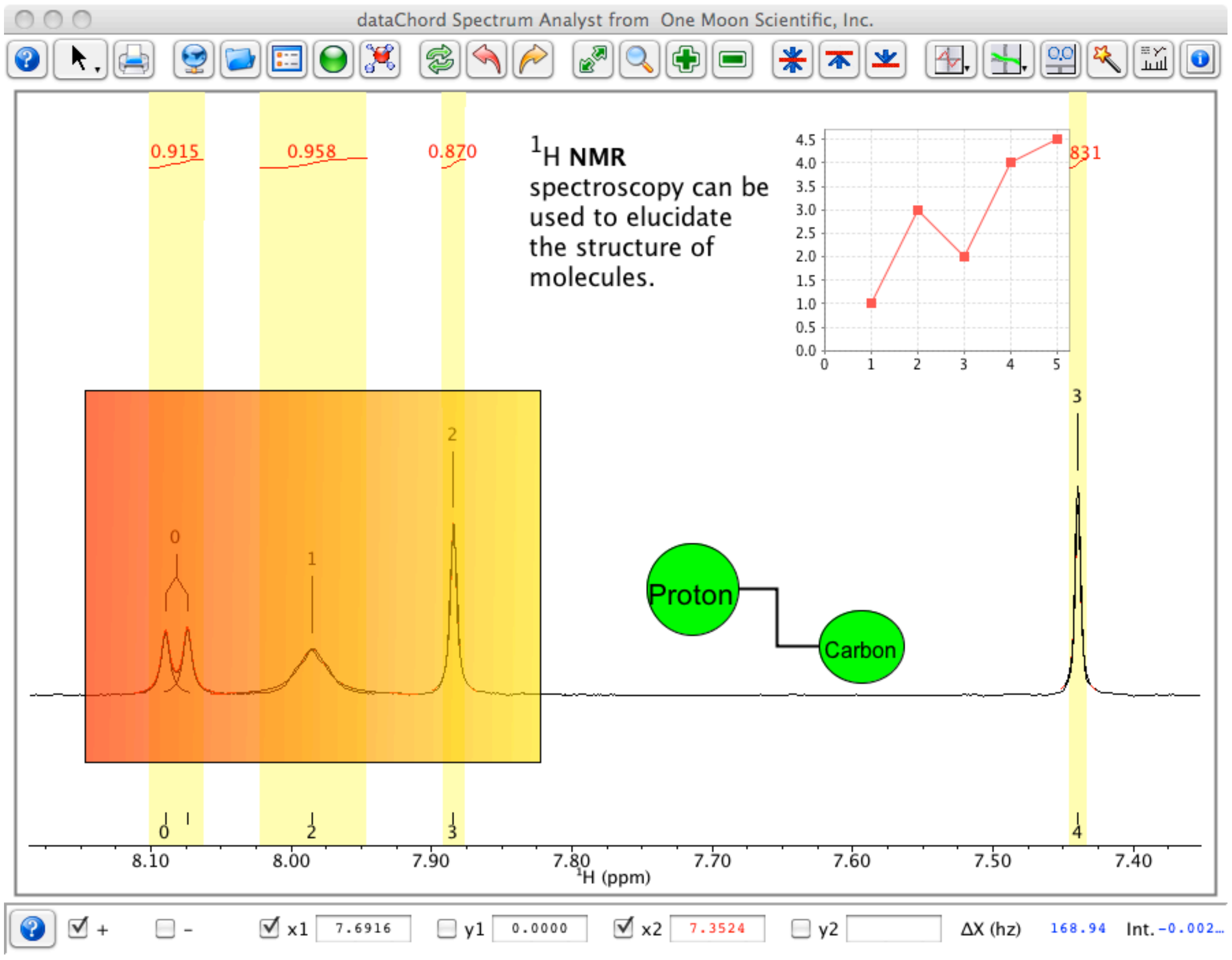
dataChord Spectrum Analyst from One Moon Scientific, Inc.
^{1}H **NMR** spectroscopy can be used to elucidate the structure of molecules.
Proton
Carbon
0.915
0.958
0.870
831
8.10
8.00
7.90
7.80
7.70
7.60
7.50
7.40
^{1}H (ppm)
x1 7.6916
y1 0.0000
x2 7.3524
y2
ΔX (hz) 168.94
Int. -0.002…

Figure 1. This figure is a screenshot of the program dataChord Spectrum Analyst which is a Java program that integrates JTcl and Swank. The primary window is used for displaying Nuclear Magnetic Resonance (NMR) spectra, but also provides for rich annotation features by the user. The NMR spectra are analyzed, in part, using tools available from the Apache Commons Math libary [ACM11]. The spectra are rendered as custom items on the Swank canvas, so multiple spectra can be rendered in various positions and orientations on the canvas. The screenshot is somewhat contrived to show various Swank canvas items: a rectangle with a transparent, gradient color, two ovals joined by a connector item, an htext item showing the user of superscript and bold text, and a chart item, implemnted using the JFreeChart library.

Much of the data analysis of dataChord Spectrum Analyst is implemented as JTcl scripts, and as a client-server program it relies heavily on the file, channel and socket capabailites of JTcl.

Canvas3D
Besides the standard Tk-like canvas, Swank includes a canvas suitable for displaying 3D objects. The implementation is at present fairly limited, but does provide the ability to draw spheres, cylinders, cones, and text. The actual 3D graphics are implemented using Java3D.

Building and Packaging
Swank is built and packaged using the same maven-based infrastructure as used by JTcl. The primary build result is a zip file that forms a "batteries included" distribution, that includes JTcl (which as discussed above includes incr Tcl, the TJC Tcl to Java Compiler, and much of Tcllib), the chart canvas item code (including JFreeChart jar files), and the canvas3d package.

Helper scripts for starting a Swank environment are included and are analogous to those described above for JTcl. Tk distributions include a program called "wish". Swank provides helper scripts called "wisk" (and wisk.bat on Windows) that start up the same type of environment that "wish" does. Also included are helper scripts, swkcon (and swkcon.bat on Windows). These start up Swank with a Swank implementation of the TkCon console [Hobbs09].

Conclusions

Together, JTcl and Swank, provide an environment for developing applications that is very similar to that of Tcl and Tk. Most programs that will run with Tcl 8.4 will run unchanged on JTcl. Swank has a greater level of differences to Tk, but provides a high level of compatibility along with additional widgets and capabilities, especially with regards to the canvas widget.

A large advantage of developing in the JTcl/Swank environment is the ability to take advantage of other libraries implemented in Java. The developer can have a high level of confidence that the combination of JTcl and Swank with other Java libraries will run unchanged on any platform with the JVM. An example of this is the program, dataChord Spectrum Analyst (Figure 1), which is written to use JTcl and Swank, and integrates in a cross-platform way libraries for math, statistics and charting.

JTcl is hosted at http://jtcl.kenai.com/ and Swank at http://swank.kenai.com/. Installers, source code, documentation, mailing lists and bug trackers are available for both projects at these sites.

References

[ACM11]
Commons Math: The Apache Commons Mathematics Library
http://commons.apache.org/math/

[DeJ05]
Incr Tcl extension for Jacl
TclJava project
http://sf.net/projects/tcljava
http://sourceforge.net/mailarchive/message.php?msg_id=1134245

[DeJ06]
TJC : A Tcl to Java Compiler
Mo DeJong
Thirteenth Annual Tcl/Tk Workshop, 2006
http://modejong.com/publications.html

[Gilb11]
JFreeChart
David Gilbert

http://www.jfree.org/jfreechart

[Hobbs09]
TkCon Project
Jeffrey Hobbs
http://tkcon.sourceforge.net/

[John04]
"From C to Java, Scientific Data Analysis with Java, Jacl and Swank"
Bruce A. Johnson
11'th Annual Tcl/Tk Conference
http://www.tcl.tk/community/tcl2004/Papers/

[Kapl02]
DustMote
http://wiki.tcl.tk/4333

[Lam97]
"Jacl: A Tcl Implementation in Java"
Ioi K. Lam, Brian Smith
Fifth Annual Tcl/Tk Workshop, 1997
http://www.usenix.org/publications/library/proceedings/tcl97/lam.html

[Lew96]
"An On-the-fly Bytecode Compiler for Tcl"
Brian T. Lewis
Fourth Annual USENIX Tcl/Tk Workshop, 1996
http://www.usenix.org/publications/library/proceedings/tcl96/lewis.html

[Lind99]
"The Java™ Virtual Machine Specification, 2nd Ed.", T. Lindholm and F. Yellin, 1999, Prentice Hall.

[Oracle2004]
http://download.oracle.com/javase/1,5.0/docs/api/overview-summary.html

[Ost10]
"Tcl and the Tk Toolkit, 2nd Ed.", John. K. Ousterhout and Ken Jones, 2010, Addison-Wesley.

[Poin07]
Aejaks Project
Tom Poindexter
http://sf.net/projects/aejaks

[Szul09]
Tcl/Tk Community Google Summer of Code 2009
Jacl Modernization Project
http://wiki.tcl.tk/23812

[WikiJVM]
JVM Languages
http://en.wikipedia.org/wiki/JVM_languages