Jacl und Tclblend verbinden Tcl und Java

Fremde Federn

Java und Tcl ergänzen sich bestens und profitieren voneinander: Ein Testskript für eine Java-Applikation ist in Tcl schneller geschrieben als in Java. Andererseits wollen Tcl-Entwickler gelegentlich Java-Pakete nutzen. Jacl und Tclblend geben ihnen diese Freiheiten. Carsten Zerbst

Die Programmiersprachen Java und Tcl sind längst feste Größen in der Welt der Compiler und Skripte. Doch auch eingefleischte Anhänger der einen Welt können kaum die Vorzüge der jeweils anderen verhehlen. Oft wäre es praktisch, in einem Tcl-Programm zusätzlich auf Java-Pakete und -Klassen zuzugreifen oder ein Java-Programm um Tcl-Scripting-Fähigkeiten zu ergänzen. Die Brücke zwischen beiden Welten schlagen Jacl und Tclblend, beide Softwarepakete des Tcljava-Projekts [1].

Jacl ist eine komplett neue Implementation des Tcl-Interpreters in reinem Java. Er führt Tcl-Skripte bis auf wenige Ausnahmen so aus, als ob sie in der normalen C-basierten Tclsh liefen. Ähnliches gibt es auch für andere Skriptsprachen wie Python (Jython, [3]) oder ECMA-Script/Javascript (Rhino, [4]). Je nach Aufgabe startet die Java-Anwendung den Interpreter mit einem Skript oder das Skript kontrolliert den Ablauf und greift auf Java-Methoden zu.

Jacl

Ioi Lam begann als Student in Cornell damit, den Tcl-Interpreter in Java neu zu schreiben. Inzwischen arbeitet vor allem Mo DeJong von Red Hat an Jacl. Bruce Johnson ergänzte mit seinem Swank [2] noch eine Tk-Variante, die er ebenfalls in reinem Java implementiert hat. Als Java-Software lassen sich Jacl und Swank über Java Webstart ausprobieren. Wer darin die Tkcon startet, hat eine komfortable Tcl-Shell frei Haus. Dazu muss eine JVM installiert sein, insbesondere für Webstart sollte es allerdings eine aktuelle Version (1.4.2) sein.

In Abbildung 1 ist die Java-Version der Tkcon zu sehen. Mit dem Tcl-Kommando »puts« hat der Benutzer zunächst einen Hallo-Welt-Text ausgegeben (blau dargestellt) und dann mit mehreren Tk-Kommandos ein weiteres Fenster mit einem Button erzeugt. Dank Swank ist das Fenster tatsächlich mit Swing- und nicht mit nativen Tk-Widgets gebaut. Sogar die Systemvariablen im »env«-Array entsprechen den Java-System-Properties, nicht den normalen Tcl-Werten.

Die Installation von Jacl und Swank gelingt am einfachsten mit der Tar-Datei von Swank [2], sie enthält bereits alle notwendigen Java-Bibliotheken sowie die Startskripte. Nach dem Auspacken startet der Aufruf »./wisk *Skriptname*« ein Skript im Interpreter, ganz wie bei »wish«. Die Swank-Distribution enthält unter anderem das Skript »swkon.tcl«, es handelt sich um die Java-Version der Tkcon (in Abbildung 1 benutzt).

Tcl und Tk auf Java-Basis statt mit C ist technisch interessant – damit sollte es sogar möglich sein, Tcl auf einem Java-fähigen Handy zum Laufen zu bringen. Weitaus wichtiger ist jedoch der Zugriff auf die jeweils andere Sprache.

Federführend

Für den Zugriff von Tcl auf Java ist ein Tcl-Paket namens »java« zuständig. Es enthält Funktionen, mit denen ein Tcl-Programm Java-Objekte und -Methoden aufrufen kann. Listing 1 zeigt ein Tcl-Skript, das die Java-Klasse in Listing 2 verwendet. Die Klasse enthält neben einfachen Getter- und Setter-Methoden (Beans-Konzept) auch mehrere Konstruktoren, statische Variablen und überladene Methoden.

Nach dem üblichen »package require java« (Zeile 3 in Listing 1) steht in Tcl eine Reihe neuer Kommandos im Namensraum »java« zur Verfügung. Eine Zusammenfassung der Befehle zeigt Tabelle 1. Damit Jacl die zusätzlichen Java-Klassen findet, setzt das Tcl-Skript in Zeile 4 einen zusätzlichen Classpath. Alternativ könnte man vor dem Start der JVM auch die globale »CLASSPATH«-Variable ändern.

Statische Klassenvariablen und Methoden

Als einfachsten Zugriff verwendet der Code statische Variablen und Methoden: In Zeile 7 liest »java::field« eine Variable der Klasse und in Zeile 10 ruft »java::call« eine Methode auf. Die Eingabe besteht aus dem Klassennamen sowie dem Variablen- oder Methodennamen und eventuellen Parametern. Danach erzeugt das Kommando in Zeile 14 eine Instanz der Demo-Klasse mit »java::new«. Der Rückgabewert enthält eine Referenz auf das neu erzeugte Java-Objekt. Der Konstruktor der Demo-Klasse (Listing 2, Zeile 13) speichert den angegebenen String in der Variablen »a«.

Für die Variable »a« implementiert die Demo-Klasse Bean-typisch so genannte Getter (Zeile 25) und Setter (Zeile 24). Die Getter und Setter sind »public« deklariert, während sich »a« dank »pri-

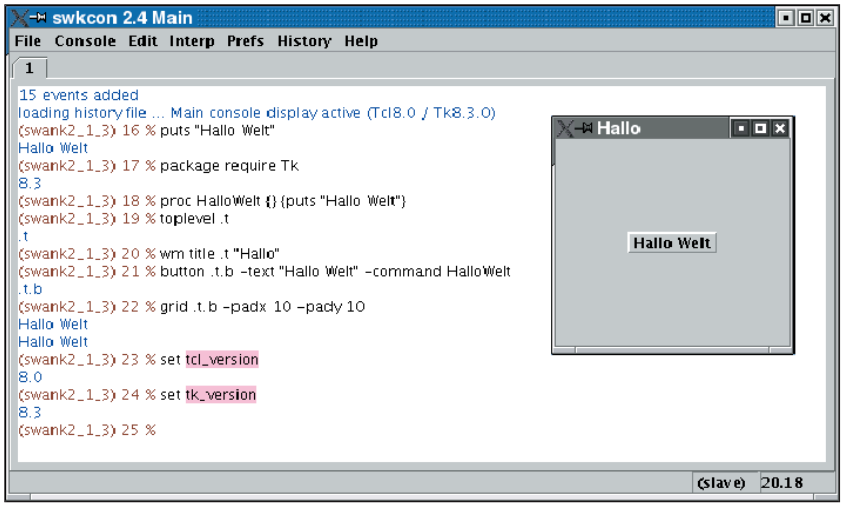

Abbildung 1: Die Tkcon läuft hier als Java-Anwendung: Interpreter und Widget-Set sind in reinem Java implementiert. Der Benutzer kann dennoch normale Tcl- und Tk-Kommandos eintippen (links) und damit zum Beispiel neue GUIs gestalten (rechts).

Tabelle 1: Wichtige Kommandos des Java-Pakets

Kommando	Aufgabe
java::new *Klassenname Argumente*	Erzeugt neue Java-Objekte
java::call *Klassenname Methodenname Argumente*	Ruft statische Methoden auf
java::field *Klasse/Objekt Feldname [Wert]*	Fragt oder setzt Objektattribute
java::instanceof *Java-Objekt Klassenname*	Wie »instanceof« in Java
java::prop *Java-Objekt Name [Wert]*	Fragt oder setzt Bean-Style-Attribute
java::bind *Java-Objekt Event-Name Skript*	Bindet ein Tcl-Skript an einen Java-Event
java::null	Gibt das Java-Objekt »null« zurück
java::isnull *Objekt*	Testet Java-Objekt gegen »null«
java::throw *Throwable*	Wirft eine Java-Exception
java::try *Skript* catch {*Klassenname Variable*} *Skript* finally *Skript*	Das Äquivalent zu Javas Try/Catch/Finally-Konstrukt
java::cast *Klassenname Java-Objekt*	Äquivalent zu Java »cast«
java::import *Klassen/Paketname*	Äquivalent zu Java »import«

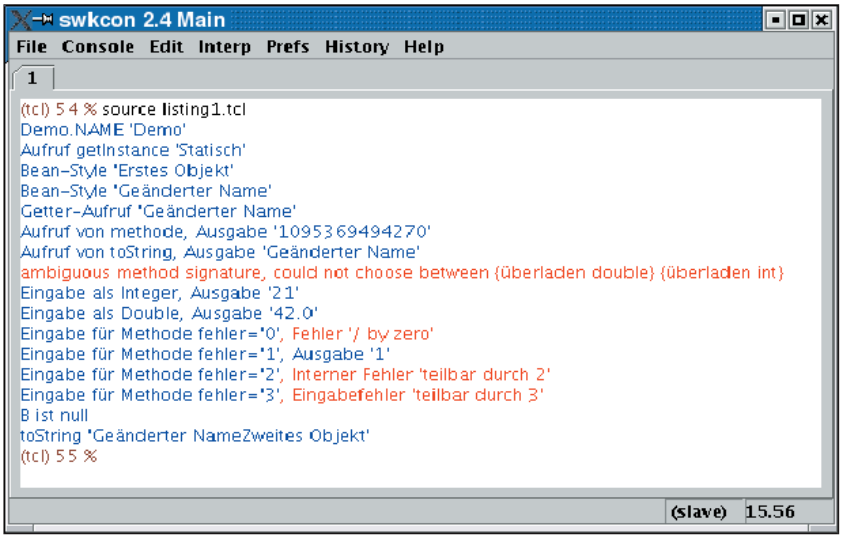

Abbildung 2: In Jacl ausgeführt zeigt das Tcl-Skript aus Listing 1 seine Ausgaben. Es benutzt die Java-Klasse »Demo« und instantiiert Objekte, ruft Methoden auf und fragt Felder ab.

vate«-Deklaration nur von der Klasse selbst lesen und ändern lässt. Für Bean-Style-Zugriffe steht unter Tcl das Kommando »java::prop« zur Verfügung, Listing 1 fragt damit den Wert ab (Zeile 17) und ändert ihn (Zeile 18).

Tcl mit Methode

Alle Java-Methoden kann das Skript Tcl-typisch mit »*Objektreferenz Methodenname Parameter*« aufrufen. Listing 1 zeigt dies anhand von »getA« in Zeile 22 und mit der Methode »methode« in Zeile 23. Ein Problem tritt jedoch bei überladenen Methoden in der Java-Klasse auf. Da Tcl keine typisierte Sprache ist, weiß Jacl nicht, ob es zum Beispiel die Ziffer 1 als String, als Integer oder als Double-Wert zu interpretieren hat. Folglich produziert Zeile 27 eine Fehlermeldung, in Abbildung 2 ist diese Meldung rot hervorgehoben (»ambigious method signature …«). Um diese Hürde zu umgehen, muss ein Tcl-Skript bei überladenen Methoden die Signatur (also den Typ der Parameter) ebenfalls angeben (Zeilen 30 und 31).

Der Beispielcode in Zeile 27 zeigt auch die Fehlerbehandlung. Die übliche Tcl-

Abbildung 3: Die Java-Anwendung »NamensSchema« in Listing 3 benutzt das Tcl-Skript aus Listing 4, um Zeichenketten nach dem vorgegebenen Namensschema umzuwandeln.

Prozedur »catch« funktioniert wie gehabt, sie erhält jedoch nur die Java-Ausgabe von »*Exception*.getMessage« und nicht die Klasse der Exception.

Wer die Fehlerarten genauer unterscheiden will, nutzt »java::try« (Zeilen 36 bis 44). Ähnlich wie in Java darf der Programmierer mehrere »catch«-Anweisungen anhängen, in denen er die erwartete Klasse der Exception (etwa »IllegalArgumentException« in Zeile 38) sowie eine Variable angibt. Die in dieser Tcl-Variablen gespeicherte Objektreferenz gibt Zugriff auf das Java-Exception-Objekt (Zeile 39). Damit hat Tcl die komplette Kontrolle über Java-Objekte.

Wer hat die Kontrolle

Mit der gezeigten Technik kontrolliert ein Tcl-Skript den Ablauf und nutzt Java-Klassen. Sinnvoll ist das vor allem, wenn eine Bibliothek nur in Java vorliegt und eine Tcl-Anwendung diese Li-

brary benötigt. Der häufigere Fall für die Tcl-Java-Kombination ist, eine kompilierte Java-Anwendung mit Scripting-Fähigkeiten auszustatten. Damit ist es möglich, Anpassungen an Kundenwünsche schnell durch ein Skript durchzuführen, angefangen von Menüs bis hin zu benutzerspezifischer Logik.

Ein typisches Problem sind Namensregeln für Objekte, etwa Rechnungen, Stücklisten oder eine CD-Sammlung. Hier kocht jeder Kunde sein eigenes Süppchen und möchte seine Regeln vollständig erfüllt sehen. Scripting erlöst den Hersteller von der Strafarbeit, für jeden Kunden eine eigene Klasse entwickeln und installieren zu müssen. Ein Skript erhält alle notwendigen Informationen, um den gewünschten Namen zu ermitteln. Die Java-Klassen implementieren dann die kundenunabhängige Logik, um etwa einen Report zu erzeugen und in einer Datei zu speichern.

Ein einfaches Beispiel ist in Listing 3 zu sehen, mögliche Namensregeln dazu sind in den Listings 4 und 5 enthalten. Der Java-Teil in Listing 3 benötigt den Namen für ein Objekt. Statt einer festgeschriebenen Regel oder eines einfachen Namenstemplate ruft es ein Tcl-Skript auf, das die Namen generiert. Dieses Skript ist schnell beim Kunden oder sogar vom Kunden selbst erstellt und einfach anzupassen.

Tcl ergänzt Java

Die Java-Klasse führt in der statischen Methode »buildName« (Zeilen 8 bis 42) ein Tcl-Skript für das angegebene Objekt aus, als Klasse für dieses Objekt dient das bekannte »Demo« aus Listing 2. Die

Listing 1: Java-Objekte in Tcl

```
01 # Benutzt die Java-Klasse "Demo" aus Listing 2
02
03 package require java
04 set env(TCL_CLASSPATH) "."
05
06 # statische Variable
07 puts "Demo.NAME '[java::field Demo NAME]'"
08
09 # statische Methode
10 set instance0 [java::call Demo getInstance
   "Statisch"]
11 puts "Aufruf getInstance '[java::prop
   $instance0 a]'"
12
13 # Objekt erzeugen
14 set instance [java::new Demo "Erstes Objekt"]
15
16 # Wert abfragen und ändern
17 puts "Bean-Style '[java::prop $instance a]'"
18 java::prop $instance a "Geänderter Name"
19 puts "Bean-Style '[java::prop $instance a]'"
20
21 # Methode aufrufen
22 puts "Getter-Aufruf '[$instance getA]'"
23 puts "Aufruf von methode, Ausgabe '[$instance
   methode]'"
24 puts "Aufruf von toString, Ausgabe '[$instance
   toString]'"
25
26 # überladene Methode
27 catch {$instance überladen 1} err
28 puts stderr $err
29
30 puts "Eingabe als Integer, Ausgabe '[$instance
   {überladen int} 42]'"
31 puts "Eingabe als Double, Ausgabe '[$instance
   {überladen double} 42.0]'"
32
33 # Fehler fangen
34 foreach wert {0 1 2 3} {
35   puts -nonewline "Eingabe für Methode
   fehler='$wert'"
36   java::try {
37     puts ", Ausgabe '[$instance fehler $wert]'"
38   } catch {IllegalArgumentException iaexp} {
39     puts stderr ", Eingabefehler '[$iaexp
   getMessage]'"
40   } catch {IllegalStateException isexp} {
41     puts stderr ", Interner Fehler '[$isexp
   getMessage]'"
42   } catch {Exception exp} {
43     puts stderr ", Fehler '[$exp getMessage]'"
44   }
45 }
46
47 # gegen null prüfen
48 set b [java::prop $instance b]
49 if [java::isnull $b] {
50   puts "B ist null"
51 }
52
53 # Objektreferenzen in Java verwenden
54 set instance2 [java::new Demo "Zweites Objekt"
   $instance]
55 puts "toString '[$instance2 toString]'"
```

Listing 2: Benutzte Java-Klasse

```
01 // Demo.java
02 import java.awt.event.*;
03
04 /**
05  * Beispielklasse für Tcl/Java-Integration
06  */
07 public class Demo {
08   public static final String NAME = "Demo";
09   private String a;
10   private Demo b;
11   private int c;
12
13   public Demo (String a) {
14     this.a = a;
15   }
16   public Demo (String a, Demo b) {
17     this.a = a;
18     this.b = b;
19   }
20   public static Demo getInstance (String a0) {
21     return new Demo(a0);
22   }
23
24   public void   setA (String a) {this.a = a;}
25   public String getA ()         {return a;}
26   public Demo   getB ()         {return b;}
27   public void   setC (int c)    {this.c = c;}
28   public int    getC ()         {return c;}
29
30   public String fehler (long a) {
31     if (a>0) {
32       if ( (a%3) == 0 ) {
33         throw new IllegalArgumentException
   ("teilbar durch 3");
34       } else if ( (a%2) == 0 ) {
35         throw new IllegalStateException
   ("teilbar durch 2");
36       }
37     }
38     return String.valueOf (1/a);
39   }
40
41   public long methode() {
42     return System.currentTimeMillis();
43   }
44   public String toString() {
45     return (b == null) ? a : (b.toString( ) +
   a);
46   }
47
48   public String überladen (int a)    {return
   String.valueOf(a/2); }
49   public String überladen (double a) {return
   String.valueOf( a ); }
50 }
```

Das Neueste

Sprachen-übergreifendes Programmieren ist nicht nur mit Tcl und Java möglich, dank Jean Luc Fontain kann ein Programmierer sogar Python-Code direkt in Tcl einbetten und Python-Methoden aufrufen [5]. Bleibt zu hoffen, dass sich dabei niemand mit den Leerzeichen verzählt – Python benutzt Einrückungen statt Klammern, um Codeblöcke zu kennzeichnen. Zur Entspannung könnte dann beruhigende Musik notwendig sein. Echte Tcl-Fans nutzen dafür das neue Snackamp [6]. Es verarbeitet viele Formate von Ogg Vorbis bis MP3, die es selbst abspielt oder als Shoutcast-Server an Streaming-Clients verfüttert.

Klienten ganz anderer Art hat Stefan Vogel mit der WebDAV-Erweiterung [7] des Tclhttpd [8] im Auge. Hiermit arbeitet der Tcl-basierte Webserver als WebDAV-Server, um Dateien plattformübergreifend auszutauschen. Viele Editoren können Files remote per WebDAV-Protokoll bearbeiten.

Wettbewerb der Komprimierer

Richard Suchenwirth hat schon viele interessante Programme im Wiki veröffentlicht. Er stellt aber auch Aufgaben an die Wiki-Leser, diesmal ist es die verlustfreie Kompression eines Schwarzweißbilds in Tcl [9]. Als Ausgangsbasis dienen einige Gif-Bilder. Das bislang beste Ergebnis von Pascal Scheffer reduziert die Größe teilweise auf ein Drittel.

Auf der GUI-Seite geht die Entwicklung weiter. Das Tile-Projekt [10] arbeitet an einer behutsamen Aktualisierung des Look&Feel von Tk. Die Version 0.4 enthält neben vielen Verbesserungen vor allem ein Combobox-Widget und lässt Tcl jetzt auf allen drei großen Plattformen modern aussehen. Einen anderen Weg geht Peter Baum mit seinem Gnocl-Paket [11]. Es benutzt GTK 2, um Widgets darzustellen, allerdings um dem Preis fehlender Tk-Kompatibilität auf der Skriptseite.

Methode muss zuerst einen Interpreter erzeugen (Zeile 13 in Listing 3). Dem Tcl-Interpreter übergibt das Java-Programm mit »setVar« das Objekt als Übergabeparameter (Zeile 16). Dazu muss es dieses Objekt in ein Objekt der Klasse »Tcl_Object« einbetten. Mit der Hilfsklasse »tcl.lang.ReflectObject« ist dies schnell erledigt. Das Objekt steht dann auf der Tcl-Seite für Zugriffe bereit.

Als Nächstes sorgt Zeile 21 dafür, dass der eingebettete Interpreter das externe Skript mit »evalFile« ausführt, Zeile 24 kopiert anschließend den Wert der Tcl-Variablen »retval« in die gleichnamige Java-Variable. Der Interpreter wird in Zeile 27 freigegeben, die Java-Garbage-Collection kann ihn danach löschen.

Interpreter-Recycling

Braucht ein Java-Programm den Interpreter sehr oft, dann ist es sinnvoller, eine Instanz zu behalten und erneut zu verwenden. Mit einem Interpreter pro Name braucht das Beispiel auf einem AMD 2600+ mit 1 GByte RAM knapp 600 Millisekunden, mit einer recycelten Interpreterinstanz reduziert sich dies auf 160 Millisekunden.

Die Listings 4 und 5 zeigen Beispiele für Namensskripte. Damit Listing 3 das gewünschte Skript auch findet, muss es »namensschema.tcl« heißen. Der erste Fall ist einfach (Abbildung 3), er nutzt die »toString«-Methode (Zeile 4) des übergebenen Objekts und hängt das aktuelle Datum an den String (Zeile 5). Die zweite Variante setzt den Namen abhängig von den Variablen »a«, »b« und »c«, wobei es nicht für alle Kombinationen gültige Namen erzeugen soll.

Listing 3: In Java eingebetteter Tcl-Interpreter

```
01 // NamensSchema.java
02 import tcl.lang.*;
03
04 public class NamensSchema {
05   /** Erzeugt einen Namen für das Objekt.
06    * Verwendet das Tcl-Skript im File
   "namensschema.tcl"
07    */
08   public static String buildName (Demo object)
   {
09     String retval = null;
10
11     try {
12       // Interpreter Object
13       Interp interp = new Interp();
14
15       // Eingabewert setzen
16       interp.setVar("object",
   ReflectObject.newInstance
17         (interp, Demo.class, object), 0);
18
19       // Skript starten
20       String skript = "namensschema.tcl";
21       interp.evalFile(skript);
22
23       // Ergebnis abfragen
24       retval = interp.getVar("retval",
   0).toString();
25
26       // Interpreter löschen
27       interp.dispose();
28     } catch (TclException tclexp) {
29       System.err.println("Fehler im Skript: "
30         + tclexp.getMessage());
31     } catch (Exception exp) {
32       if (exp instanceof NameSchemaViolation)
   {
33         System.err.println("Objekt verletzt
   Namensschema: "
34           + exp.getMessage());
35       } else {
36         exp.printStackTrace();
37         System.err.println("Es trat ein Fehler
   auf: "
38           + exp.getMessage());
39       }
40     }
41     return retval;
42   }
43
44   public static void main (String[] args) {
45     Demo a = new Demo("Einfach");
46     a.setC(-12);
47     System.out.println("Name für '" + a +
   "'");
48     System.out.println("\t\t" +
   NamensSchema.buildName(a));
49
50     a = new Demo("C==0");
51     a.setC(0);
52
53     System.out.println("Name für '" + a + "'"
   );
54     System.out.println("\t\t" +
   NamensSchema.buildName(a));
55
56     a = new Demo("Verkettet", new
   Demo("Basis"));
57     a.setC(42);
58     System.out.println("Name für '" + a + "'"
   );
59     System.out.println("\t\t" +
   NamensSchema.buildName(a));
60   }
61 }
```

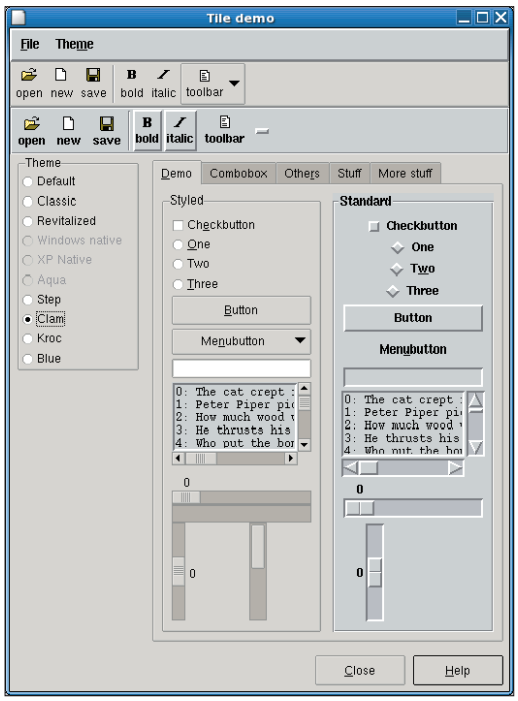

Abbildung 4: Tk wird mit dem Tile-Paket Theme-fähig. Das Clam-Thema verleiht Tk-Programmen einen modernes Aussehen.

Die Werte der Variablen extrahiert das Skript per Getter-Methode aus dem übergebenen Java-Objekt (Zeilen 11, 21, 23 und 25). Als Startwert für den Rückgabewert beginnt Listing 5 mit einem leeren String (Zeile 9). Abhängig von »$c« hängt es danach den Buchstaben »A« an (Zeile 13), wirft eine Exception (Zeile 15) oder fügt den Buchstaben »B« zusammen mit der Variablen »$c« an. Damit das Skript die Exception werfen kann, muss eine gleichnamige Java-Klasse existieren:

```
public class NameSchemaViolation
      extends Exception {
}
```

Diese Klasse braucht keine eigenen Methoden zu implementieren, es genügt, wenn sie von »Exception« erbt.

Soll ein Skript Oberflächen erstellen, wäre dies entweder direkt mit den Klassen von Swing oder über eigene Bibliotheken möglich. Um bestehende Skripte zu verwenden oder schnell eine Oberfläche zu bauen, bietet sich Bruce Johnsons Swank [2] an. Es stellt die meisten Tk-Befehle auch unter Jacl bereit, setzt allerdings Swing als Basis ein.

Tclblend

Neben Jacl gibt es mit Tclblend [1] noch eine weitere Tcl-Java-Kopplung. Tclblend verbindet den in C implementierten Tcl-Interpreter und die JVM über ihre JNI-Schnittstelle (Java Native Interface). Anders als bei Jacl steht auf diese Weise der volle Umfang von Tcl und Tk sowie beliebigen Erweiterungen zur Verfügung.

Vor dem Einsatz von Tclblend steht jedoch ein größerer Kompilierungsaufwand, da Tcl mit Thread-Unterstützung übersetzt sein muss, hinzu kommt noch Tclblend selbst. In den meisten Fällen ist Jacl die einfachere Alternative, vor allem kann ein Java-Programm damit auch neue Interpreter erzeugen. Die Java-spezifischen Tcl-Funktionen sind bei Jacl und Tclblend identisch, beide verwenden das »java«-Paket.

Gut für Spezialfälle

Die Kopplung von Tcl und Java ist zwar kaum der Default-Weg, um neue Anwendungen zu entwickeln. Die meisten Programmierer haben ja genug damit zu tun, sich in einer Sprache sicher zu bewegen. Dennoch hat die Kombination ihren Reiz. Sie öffnet Tcl den Zugriff auf Bibliotheken, die nur in Java verfügbar sind, etwa Apaches Batik. Java-Anwendungen können vom Scripting kundenspezifischer Teile gewinnen. *(fjl)* ■

Infos

[1] Tcl/Java-Integration mit Jacl und Tclblend: [http://tcljava.sourceforge.net]
[2] Swank, ein Tk-Klon in reinem Java: [http://www.onemoonscientific.com/swank/]
[3] Jython, ein Python-Interpreter in Java: [http://www.jython.org]
[4] Rhino, ein Javascript-Interpreter in Java: [http://www.mozilla.org/rhino/]
[5] Tclpython, Integration von Tcl und Pyhton: [http://jfontain.free.fr/tclpython.htm]
[6] Tcl-basierter Sound-Player Snackamp: [http://snackamp.sourceforge.net]
[7] WebDAV: [http://www.vogel-nest.de/wiki.php/Main/WebDAV]
[8] Tclhttpd, in Tcl programmierter Webserver: [http://www.tcl.tk/software/tclhttpd/]
[9] Schwarzweißgrafiken in Tcl komprimieren: [http://wiki.tcl.tk/12314]
[10] Tile, Themeing-Enginge für Tk: [http://tktable.sourceforge.net/tile/]
[11] Gnocl: [http://www.dr-baum.net/gnocl/]

Der Autor

Carsten Zerbst arbeitet bei Atlantec an einem PDM-System für den Schiffbau. Daneben beschäftigt er sich mit dem Einsatz von Tcl/Tk.

Listing 4: Einfaches Namensskript für Listing 3

```
01 # Einfache Namensregel,
02 # Name in Kleinbuchstaben + Datum
03
04 set retval [string tolower [$object toString]]
05 append retval -[clock format [clock seconds] -format
   "%Y%m%d"]
```

Listing 5: Komplexeres Namensskript für Listing 3

```
01 # Komplexe Namensregel
02 # C < 0  --> A
03 # C == 0 --> Fehler
04 # C > 0  --> B+ $anzahl
05 # wenn b, dann Name zuerst
06
07 package require java
08
09 set retval " "
10
11 set c [$object getC]
12 if {$c < 0} {
13   append retval A
14 } elseif {$c == 0} {
15   set exp [java::new NameSchemaViolation
     "Wert C ist 0"]
16   java::throw $exp
17 } else {
18   append retval "B$c"
19 }
20
21 set b [$object getB]
22 if {[java::isnull $b]} {
23   append retval [$object getA]
24 } else {
25   append retval [$b toString]/[$object getA]
26 }
27
28 # für Debugging:
29 puts $retval
```