Script-Programmierung Tcl/Tk

Manuel Caroli

8. April 2004

Inhaltsverzeichnis

1 Einleitung

Diese Proseminar-Arbeit soll einen kurzen Überblick über Script-Programmierung am Beispiel von Tcl/Tk geben. Zunächst werden die geschichtlichen Hintergründe betrachtet, um dann die von John K. Ousterhout Ende der achtziger Jahre entwickelte Script-Sprache Tcl näher zu erläutern. Schließlich wird das dazu gehörige Toolkit Tk betrachtet sowie die Möglichkeiten, die dadurch eröffnet werden.

2 Was ist eine Script-Sprache

Eine Script-Sprache ist eine Programmiersprache, die nicht kompiliert, sondern interpretiert wird. Das bedeutet, die Script-Dateien sind grundsätzlich im Textformat verfügbar und werden so einem Interpreter (meist einer „shell“) übergeben. Script-Programmierung ist vor allem in der Unix/Linux-Welt weit verbreitet (bash, csh, etc.). Aber auch unter MS-DOS werden Dateien mit der Endung *.bat grundsätzlich von der Shell als Script-Dateien ausgewertet. Hierbei hat jede Shell / jeder Interpreter seine eigene Syntax und seine eigenen Regeln. Zum Ausführen von Tcl-Scripts gibt es die tclsh (Tool Command Language Shell), zum Ausführen von Tk-Skripts die wish (Windowing Shell).

3 Historisches

Script-Sprachen gibt es bereits seit den sechziger Jahren. Eine der ersten dieser Script-Sprachen war JCL (Job Control Language). JCL ermöglichte die Kommunikation mit einem frühen IBM-Betriebssystem. Diese Kommunikation war speziell auf die Steuerung von Jobs ausgerichtet.

In den siebziger Jahren tauchten die Unix-Shells auf und mit ihnen die Möglichkeit immer gleiche Konfigurationsabläufe – z. B. beim Start – per Script-Dateien zu automatisieren. Diese Shells stellten, ähnlich wie JCL vor allem Interfaces zur Kommunikation mit dem Betriebssystem dar, wobei es hier bereits Kontrollstrukturen wie `while` oder `if` gab und bidirektionale Kommunikation mit Betriebssystem-Kommandos möglich war, d. h. Kommandos können Argumente übergeben bekommen und Werte zurückliefern.

Ende der achtziger Jahre erschienen dann Tcl und Perl. Hier waren die Möglichkeiten für den Programmierer noch ausgereifter als in den Unix-Shells, d. h. es standen weitere Möglichkeiten offen, die man eigentlich von Systemprogrammiersprachen wie C kannte, z. B. Referenzen, Prozesse, Prozeduren/Modularisierung etc.

In den neunziger Jahren wurde dann schließlich mit Python, Visual Basic und JavaScript eine neue Klasse von Script-Sprachen kreiert, die sogar bis hin zur Objektorientierung moderne Konzepte verwirklichen.

Die Entwicklung von Tcl begann in den achtziger Jahren an der University of California in Berkeley. Hier stellten John K. Ousterhout und sein Team fest, dass sie viel Zeit darauf verwendeten um Kommandosprachen auf spezielle Tools anzupassen. Aus

dieser Not heraus begann die Entwicklung einer neuen „Tool Command Language“ , die universeller sein sollte als die bis dahin verfügbaren Script-Sprachen.

4 Die Sprache Tcl

4.1 Das Kommando-Konzept

Ein Tcl-Script besteht aus einer Reihe von Kommandos. Jedes einzelne Kommando besteht seinerseits aus dem Kommandonamen, z. B. `set` und aus einer Reihe von String-Argumenten. Die Anzahl und Art der Argumente ist von Kommando zu Kommando unterschiedlich.

4.2 Das Beispiel fac

```
proc {fac} {x} {
  if {$x <= 1} {return {1}}
    {expr {$x * [fac [expr {$x-1}]]}}
  }
```

Durch das Kommando `proc` wird hier ein neues Kommando `fac` eingeführt, das ein Argument erhält und die Fakultät dazu berechnet. Das Kommando `proc` bekommt hier drei Argumente: Das erste Argument ist der Name des zu erzeugenden Kommandos. Das zweite Argument enthält die Argumentliste, die das zu erzeugende Kommando erhalten soll. Das dritte Argument ist wiederum ein Tcl–Skript. Dieses wird ausgeführt, wenn das erzeugte Kommando aufgerufen wird. Alle Argumente werden in Form von Strings übergeben. Das neu erzeugte Kommando `fac` lässt sich nicht mehr von bereits vorher vorhanden Kommandos unterscheiden.

An dieser Stelle wird klar, dass Tcl trotz dieses statisch erscheinenden „Kommando-Argument-Argument...–Konzeptes“ sehr flexibel ist.

Das Kommando `if` ist selbstverständlich auch – wie alle anderen Kontrollstrukturen – nach diesem Schema aufgebaut. `if` erwartet zwei oder drei Argumente. Das erste Argument ist ein „auswertbarer“ Ausdruck, das zweite und das dritte Argumente sind jeweils Tcl-Skripte, wobei das zweite ausgeführt wird, falls der Ausdruck im ersten Argument nach Auswertung 1 ergibt. Andernfalls wird das Skript im dritten Argument ausgeführt.

Ein „auswertbarer“ Ausdruck ist ein Ausdruck, der nach den Regeln der Mathematik ausgewertet werden kann. `5<3` oder `3*8 + 4` wäre solch ein Ausdruck, wohingegen `hallo` kein auswertbarer Ausdruck ist.

Das Kommando `expr` berechnet auswertbare Ausdrücke:

```
expr 3*8 + 4
```

Ausgabe:

```
28
```

Selbst das Kommando `expr` folgt dem Schema *Kommando Argument Argument...*, obwohl es hier auf den ersten Blick nicht so aussieht. Bei `expr` verhält es sich folgendermaßen: `expr` erwartet beliebig viele Argumente, aber mindestens eines. Bei der Auswertung werden zunächst alle Argumente konkateniert, und dann wird der so entstandene Ausdruck wenn möglich ausgewertet. In obigem Beispiel wäre `5<1` ein Argument und `3*8 + 4` wären drei Argumente

4.3 nur Kommandos und Strings

Es gibt in Tcl zwei Grundsätze:

1. Ein Skript ist eine Folge von Kommandos
   Wie oben beschrieben wird in Tcl alles (sogar Kontrollstrukturen) durch Kommandos beschrieben. Einem Kommando kann eine Reihe von Argumenten übergeben werden. Das Kommando und die Argumente werden durch „Worttrenner“ unterschieden. Worttrenner sind z. B. Leerzeichen. Die Kommandos werden durch neue Zeilen voneinander getrennt. Das bedeutet, dass ein Kommando aus einer Folge von Strings besteht, die durch Leerzeichen getrennt sind, ein Skript ist eine Folge von Kommandos die ihrerseits durch neue Zeilen getrennt sind.

2. Es gibt nur einen Datentyp: String
   Jedes Kommando bekommt eine Folge von Strings übergeben. Demzufolge genügt es, dass Strings in Tcl der einzige Datentyp ist. Es kann trotzdem zwischen Zeichenketten und Zahlenwerten unterschieden werden. Das geschieht dadurch, dass ein Kommando an einer bestimmten Stelle entweder einen Zahlenwert oder eine Zeichenkette erwartet. Erfüllt hier der Zahlenwert, der ja auch als Zeichenkette übergeben wird, nicht bestimmte Bedingungen, das heißt kann er nicht als arithmetischer Ausdruck ausgewertet werden, so ergibt dies einen Fehler.

Um nun die Übergabe von Argumenten und die Auswertung von Kommandos steuern zu können gibt es die Möglichkeiten von Quoting und Substitution, die im nächsten Abschnitt näher betrachtet werden.

4.4 Quoting und Substitution

Zwei wichtige Punkte bei Tcl sind „Quoting“ und „Substitution“. Da in Tcl den Kommandos nur unstrukturierte String-Argumente übergeben werden können muss es irgendwie möglich sein die Ausführung genauer zu steuern. Hierzu gibt es die Möglichkeit von Quoting und Substitution. Quoting bedeutet, dass der übergebene String nicht verändert wird, während Substitution bedeutet, dass er durch irgend etwas ersetzt wird.

4.4.1 Substitution

Es gibt in Tcl drei Arten von Substitution: $–Substitution, Substitution mit \ und Substitution mit `[]`.

- $–Substitution
  Geht einer Variablen ein $-Symbol voran, so wird diese Variable (inklusive $-Symbol) durch ihren Wert ersetzt. Diese Ersetzung findet grundsätzlich statt, es sei denn, sie wird durch Quoting aufgehoben. Dazu mehr im nächsten Abschnitt.

  Beispiel:
  ```
  set x 10
  expr $x+1
  ```

  Ausgabe:
  11

- Substitution mit \
  Mit Hilfe des \ ist es möglich auch Worte die Sonderzeichen enthalten zu konstruieren. Zu den Sonderzeichen zählen hier die Worttrenner (Leerzeichen, neue Zeile, etc.), das oben beschriebene $–Symbol u.a.

  Beispiel:
  ```
  set x 1\ EUR\ kostet\ 1,2226\ \$\n1\ \$\ kostet\ 0,8179\ EUR
  ```

  Ausgabe:
  1 EUR kostet 1,2226 $
  1 $ kostet 0,8179 EUR

  Es finden hier folgende Ersetzungen statt:
  \„leer“ wird zu „leer“
  \$ wird zu $
  \n wird zu „neue Zeile“

- Substitution mit []
  Alles was innerhalb von [] steht wird grundsätzlich zunächst ausgewertet und durch das Ergebnis ersetzt.

  Beispiel:
  ```
  set x [expr 1+2]
  ```

  Ausgabe:
  3

4.4.2 Quoting

Mit Quoting können die oben beschriebenen Substitutionen zum Teil aufgehoben werden. Es gibt in Tcl zwei Arten von Quoting: Quoting mit geschweiften Klammern und Quoting mit Anführungszeichen.

- Quoting mit geschweiften Klammern
  Beim Quoting mit geschweiften Klammern wird der Inhalt der geschweiften

Klammern **unverändert übergeben**, das bedeutet es findet **keine** der oben beschriebenen Substitutionen statt.

Beispiel:
```
set newString {1 EUR kostet 1,2226 $
1 $ kostet 0,8179 EUR}
```

Ausgabe:
```
1 EUR kostet 1,2226 $
1 $ kostet 0,8179 EUR
```

- Quoting mit Anführungszeichen
  Beim Quoting mit Anführungszeichen werden alle Worttrenner ignoriert. Die oben beschriebenen Substitutionen finden nach wie vor statt. Also müsste obiges Beispiel dann folgendermaßen aussehen:

  ```
  set newString "1 EUR kostet 1,2226 \$\n1 \$ kostet 0,8179 EUR"
  ```

  oder

  ```
  set newString "1 EUR kostet 1,2226 \$
  1 \$ kostet 0,8179 EUR"
  ```

  Ausgabe in beiden Fällen:
  ```
  1 EUR kostet 1,2226 $
  1 $ kostet 0,8179 EUR
  ```

4.5 quoting hell

Nun, da sowohl Substitution als auch Quoting bekannt sind, ist es an der Zeit sich ein paar Gedanken zu machen, was geschieht, wenn man größere Skripte schreibt. Dann kommt es unweigerlich (je nach Programmierstil früher oder später) dazu, dass ein Skript extrem unübersichtlich wird und man vor lauter Klammern nicht mehr erkennen kann, was das Skript genau tut. Tcl ist eine Sprache, die sehr schnell zu Unübersichtlichkeit neigt wenn man in größeren Skripten mit Substitution und Quoting arbeitet, was man ja unweigerlich muss. Es wird dann auch sehr schwierig Fehler zu finden. Diese Tatsache wird treffend mit den englischen Worten „quoting hell“ bezeichnet. „quoting hell“ ist ein großer Nachteil von Tcl und man muss sich dessen klar sein, wenn man plant größere Skripte in Tcl zu realisieren.

4.6 Scoping

In Tcl gibt es *globale* und *lokale* Variablen. Auf globale Variablen kann von überall aus zugegriffen werden, während lokale Variablen nur in dem Scope, in dem sie erzeugt wurden verfügbar sind.

Allerdings haben verschiedene Scopes verschiedene Namensräume und standardmäßig wird immer auf den eigenen Scope zugegriffen. Das heißt, falls es eine globale Variable A gibt und in einer Unterprozedur gibt es eine lokale Variable A, so kann von der Unterprozedur nur auf das lokale A zugegriffen werden. Falls es kein lokales A gibt

kann trotzdem nicht auf das globale A zugegriffen werden. Man muss globale Variablen immer zunächst „lokal verfügbar“ machen, bevor sie verwendet werden können.

Dies geschieht mit dem eingebauten Kommando `global`. `global x y` macht beispielsweise die globalen Variablen `x` und `y` innerhalb der Prozedur in der das Kommando steht verfügbar. Falls es vorher keine globalen Variablen `x` und `y` gab, so werden diese erzeugt.

Beispiel:

```
proc makeA {} {
  global A
  set A 1
}
```

Hier wird die globale Variable `A` auf 1 gesetzt. Falls sie vorher nicht existierte wird sie zuerst erzeugt.

Ein mächtigeres Kommando als `global` ist `upvar`. Mit `upvar` kann, ähnlich wie bei `global`, eine Variable aus einem anderen Scope verfügbar gemacht werden. Der Unterschied ist, dass bei `upvar` gewählt werden kann, in welchem Scope sich diese Variable befindet. Das heißt `upvar` bekommt ein zusätzliches Argument mit, das diesen Scope bestimmt. Standardwert ist 1, d.h. die Variablen aus dem „nächsthöheren Scope“ werden verfügbar gemacht. Der nächsthöhere Scope ist derjenige, aus dem die aktuelle Prozedur aufgerufen wurde. Ein Wert n bedeutet, dass auf den „n-höheren“ Scope zugegriffen wird. Auf diese Weise kann bei geschachtelten Funktionsaufrufen gesteuert werden, an welcher Stelle der Schachtelung die Variable gespeichert werden soll.
Ein spezielles Argument ist `#0`. `#0` bedeutet den höchsten, also den globalen Scope.

Beispiel:

```
proc makeA {} {
  upvar #0 A a
  set a 1
}
```

Die globale Variable `A` wird hier unter dem Namen `a` verfügbar gemacht. Bei geschachtelten Funktionsaufrufen könnte man `#0` durch eine bestimmte positive Zahl ersetzen und so erreichen, dass die Variable `A` aus dem dadurch beschriebenen Scope hier als `a` verfügbar ist. Beispiele hierzu sind sehr komplex und würden den Rahmen dieser Arbeit sprengen.

Beispiel (`fac` mit Referenzen):

```
proc fac x {
  upvar fac$x res
  set res 1
  for {set i $x} {$i > 0} {incr i -1} {
    set res [expr $res * $i]
    }
  return $res
}
```

Hier wird zunächst eine Referenz (`res`) auf die Variable `fac$x` (wobei `$x` natürlich durch den entsprechenden Wert ersetzt wird) erzeugt. Falls `fac$x` noch nicht existierte wird es neu erzeugt. Dann wird `res` (und somit auch `fac$x`) auf 1 gesetzt. In der For-Schleife wird der eigentliche Wert der Fakultät ausgerechnet und in `res` – und somit auch in `fac$x` – gespeichert. Schließlich wird das Ergebnis zurückgegeben. Ruft man also dieses Kommando `fac` auf, so erhält man das Ergebnis als Rückgabewert; außerdem wird es in der Variablen `fac$x` im aktuellen Scope gespeichert. Ein Aufruf von `fac 4` hätte beispielsweise den Effekt, dass die Variable `fac4` den Wert *24* zugewiesen bekäme.

4.7 Automatisierte Variablenerzeugung

Zur Verdeutlichung der Möglichkeiten, die Quoting und Substitution zusammen mit dem Scoping in Tcl bieten hier ein kleines Beispiel in dem auf recht elegante Weise Variablen automatisch erzeugt und initialisiert werden.

```
proc makeVar {trunk init start end scope} {
  for {set i $start} {$i <= $end} {incr i} {
    upvar [expr $scope+1] $trunk$i a
    set a $init
  }
}
```

Das neu erzeugte Kommando `makeVar` erwartet einen Stamm, einen Initialisierungswert einen Startwert und einen Endwert und erzeugt dazu die entsprechenden neuen Variablen im gewünschten Scope.
Das Kommando `makevar a 0 0 10 0` erzeugt beispielsweise im aktuellen Scope die Variablen a0, a1, ... a10, die alle mit 0 initialisiert sind.

4.8 Fehler und Ausnahmen

Es gibt in Tcl eine Fehler- und Ausnahmebehandlung. Hierbei wird zwischen Fehlern und Ausnahmen unterschieden.

4.8.1 Fehlerbehandlung

Fehler können in Tcl mit `error` geworfen und mit `catch` gefangen werden.

`error` erwartet bis zu 3 Argumente: Das erste ist die Fehlermeldung, die bei Auftreten dieses Fehlers ausgegeben wird. Das zweite Argument wird in der globalen Variablen `errorInfo`, das dritte in der globalen Variablen `errorCode` gespeichert.

`catch` erwartet bis zu 2 Argumente. Das erste ist ein Tcl-Skript, das ausgeführt wird. Tritt während der Ausführung dieses Tcl-Skripts ein Fehler auf, so wird er von `catch` gefangen und – falls angegeben – in der optional als zweites Argument angegebenen Variablen gespeichert. Falls ein Fehler auftritt gibt `catch` 1 zurück, ansonsten 0.

4.8.2 Ausnahmebehandlung

Während Fehler nur eine spezielle Art von Ausnahmen sind, gibt es in Tcl noch weitere Ausnahmen, wie `break`, `continue` und `return`. Diese unterscheiden sich in einigen Punkten von Fehlern: Die globalen Variablen `errorInfo` und `errorCode` werden hier nicht gesetzt. Ausnahmen dieser Art finden recht häufig Anwendung in Schleifen, wo sie genutzt werden um die Schleife abzubrechen (`break, return`) oder weiter auszuführen (`continue`).

4.9 Weitere Kommandos

4.9.1 Eingebaute Kommandos

- Mathematische Operationen
  In bezug auf die mathematischen Operationen ist Tcl stark an C angelehnt. Das war eine absichtliche Designentscheidung, da den meisten Programmierern die Syntax von C bekannt ist. Es sind hier prinzipiell alle mathematischen Operationen verfügbar, die man erwarten würde, wie z. B. die Grundrechenarten, Boolesche Operationen, trigonometrische Funktionen etc.

- Stringmanipulation
  Auch an Kommandos zur Stringmanipulation ist in Tcl alles nötige verfügbar.

- Listen
  Listen stellen in Tcl eine Basisdatenstruktur dar. Listenelemente werden einfach durch white spaces (Leerzeichen oder Tabs) voneinander getrennt. Dadurch sind sie sehr einfach zu handhaben. Sehr praktisch ist das Kommando `foreach`, das ein Skript für alle Listenelemente ausführt.

- Kontrollstrukturen
  Desweiteren gibt es in Tcl alle üblichen Kontrollstrukturen wie `if`, `while`, `for` etc. Diese Kontrollstrukturen sind ganz normale vordefinierte Kommandos und können von der Shell nicht von gewöhnlichen oder gar selbst definierten Kommandos unterschieden werden.

- Kommentare
  Kommentare werden in Tcl durch das Zeichen # eingeleitet. # ist übrigens ebenfalls ein Kommando, das allerdings alle Argumente ignoriert und nichts tut.

4.9.2 Externe Programme

Da die `tclsh` als Shell auf dem Betriebssystem aufsetzt, können auch alle möglichen externen Programme aufgerufen werden. Unter Windows hätte beispielsweise die Zeile `dir c:\\` den selben Effekt wie `dir c:\` in der MS-DOS-Eingabeaufforderung. Unter Linux hätte z. B. `gcc --ver` den selben Effekt wie in jeder anderen Shell.

5 Tk

Tk ist eine Erweiterung von Tcl, die auf sehr einfache Art und Weise das Erstellen von grafischen Benutzeroberflächen ermöglicht. Tk (abgekürzt für **T**ool**k**it) wurde von John K. Ousterhout etwa ein Jahr nach Tcl veröffentlicht und seitdem wird Tcl vorwiegend gemeinsam mit Tk unter der Bezeichnung „Tcl/Tk “ verwendet. Wie für Tcl die `tclsh`, so gibt es für Tk die `wish` (Windowing Shell). In der `wish` können zusätzlich zu allen Tcl-Kommandos auch die Tk-spezifischen Kommandos ausgeführt werden. Außerdem zeigt die `wish` ein kleines Fenster an, in dem die Auswirkungen der ausgeführten Kommandos sofort sichtbar werden. Im Folgenden werden die Möglichkeiten und Besonderheiten von Tk näher betrachtet.

5.1 Widgets

„Widget “ ist der Oberbegriff für Komponenten der Benutzeroberfläche. Das können Buttons, Menüs, Listboxes etc. sein. Diese Widgets sind über ihre Namen hierarchisch geordnet. Das „main-Widget “ hat immer den Namen „.“Die Namen der weiteren Widgets beginnen immer mit einem „. “, wodurch symbolisiert wird, dass diese Widgets Unterwidgets des main-Widgets sind. Ein solches Widget kann seinerseits wieder Unterwidgets haben. Der Name solcher „Unterunterwidgets “ beginnt mit einem „. “ gefolgt vom Namen des Oberwidgets, wiederum gefolgt von einem „. “ worauf dann der eigentliche Widgetname folgt. Das Widget mit dem Namen `.a.b` wäre also ein Unterwidget des Widgets mit dem Namen `.a`, was wiederum ein Unterwidgets des main-Widgets ist. Der Name eines Widgets wird bei dessen Erzeugung zugleich als Kommando registriert. Das heißt in obigem Beispiel wäre `.a.b` zugleich als Kommando registiert mithilfe dessen das Widget `.a.b` angesteuert und konfiguriert werden kann.

Beispiel:

```
button .top -text "Top button"
button .bottom -text "Bottom button"
pack .top .bottom
```

Das Kommando `button` ist ein Tk-Kommando, das Buttons erzeugt.
Das Kommando `pack` ruft den Geometriemanager *Packer* auf (vgl. nächster Abschnitt) um die beiden Buttons zu zeichnen.
Mit dem in der ersten Zeile erzeugten Kommando `.top` können nun beispielsweise – wie hier in der letzten Zeile – Eigenschaften des Widgets `.top` verändert werden.

```
.top configure -relief sunken
```

5.2 Geometriemanager

Um ein Widget nach seiner Erzeugung auch sichtbar zu machen benötigt man einen Geometriemanager. Es werden an dieser Stelle nur die zwei wichtigsten betrachtet: *Packer* und *Placer*

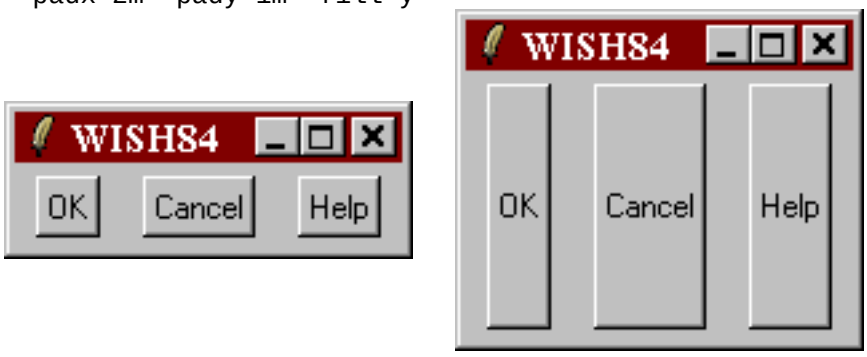
5.2.1 Packer

Der Packer ist der Standard-Geometriemanager in Tk. Er wird mit dem Kommando `pack` aufgerufen und erhält die zu zeichnenden Widgets und verschiedene Optionen als Argumente. Der große Vorteil des Packer ist seine extreme Flexibilität. Erhält er keine weiteren Optionen, so ordnet er die Widgets nach default-Werten im Fenster an. Die Optionen dienen zur Steuerung dieser Anordnung. So kann man z. B. die Widgets nebeneinander anstatt untereinander anordnen lassen, die Abstände festlegen oder dafür sorgen, dass die Widgets immer das Fenster ganz ausfüllen. Die genaue Größe auf Pixelebene zu berechnen übernimmt der Packer vollautomatisch.

Somit ist der Packer ein sehr einfach zu handhabendes Werkzeug, mithilfe dessen grafische Oberflächen gestaltet werden können.

Beispiel:

```
pack .ok .cancel .help -side left \
  -padx 2m -pady 1m -fill y
```



`.ok`, `.cancel` und `.help` sind drei bereits erzeugte Buttons. Die Option `-side left` sorgt dafür, dass die drei Buttons nebeneinander von links nach rechts angeordnet werden. Der \ hebt per \–Substitution die „neue Zeile“ auf. Die Optionen `-padx 2m` und `-pady 1m` bestimmen den horizontalen (hier 2 mm) und den Vertikalen (hier 1 mm) Abstand zum nächsten Widget bzw. zum Rand. Die Option `-fill y` lässt die Buttons in vertikaler Richtung den gesamten freien Platz ausfüllen.

5.2.2 Placer

Der Placer wird mit dem Kommando `place` aufgerufen. Auch er erhält die zu zeichnenden Widgets als Argumente. Allerdings erhält er keine Optionen zur Anordnung dieser Widgets sondern erwartet die statischen Pixelwerte für Position und Größe der einzelnen Widgets. Somit ist der Placer weit weniger flexibel und komfortabel als der Packer und findet daher auch seltener Verwendung.

Beispiel:

```
button .b -text "OK"
place .b -x 10 -y 10 -width 100
```

5.3 Ereignisbehandlung

Da es bei grafischen Benutzeroberflächen im Allgemeinen nötig ist Ereignisse behandeln zu können (event management), ist dies auch in Tk möglich. Es können hier Ereig-

nisse an Tcl-Skripts „gebunden“ werden, das heißt, falls ein Ereignis auftritt, wird das zugehörige Tcl-Skript ausgeführt. Dieses Binden von Ereignissen an Skripte geschieht durch das Kommando `bind`. `bind` erwartet drei Argumente: zunächst das Widget, auf das sich die Bindung bezieht, dann das Ereignis und schließlich das auszuführende Skript.

Beispiele:

```
bind .b <Button-1> {...}
bind . <Any-KeyPress> {puts Taste %K"}
bind all <Motion> {puts "Mausposition (%x,%y)"}
```

5.4 Kommunikation zwischen Anwendungen

Eine weitere Fähigkeit von Tcl/Tk ist die Möglichkeit der anwendungsübergreifenden Kommunikation. Zu diesem Zweck gibt es das Kommando `send`.

Die Ursache diesen Konzepts ist, dass John K. Ousterhout die Möglichkeit schaffen wollte große monolithische Programme durch viele kleine spezialisierte Programme zu ersetzen, die in ihrer Gesamtheit den selben Funktionsumfang zur Verfügung stellen. Hierzu ist es natürlich nötig, dass die Programme untereinander kommunizieren können, was in Tcl durch `send` geschieht.

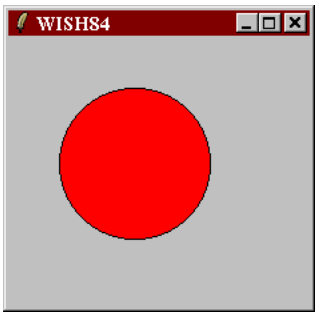
`send` ist synchronisiert, das bedeutet, dass während das `send`-Kommando etwas tut alle anderen Prozesse warten. Allerdings antwortet `send` dennoch auf andere Aufrufe von `send`, so dass es nicht zu einem Deadlock kommen kann.

5.5 Canvas

Ein Canvas ist ein Widget, das eine Zeichenoberfläche anzeigt, auf welcher verschiedenartige grafische Objekte angezeigt werden können. Hierbei handelt es sich um sogenannte grafische Primitive wie z. B. Rechtecke, Kreise, Linien etc.

Beispiel:

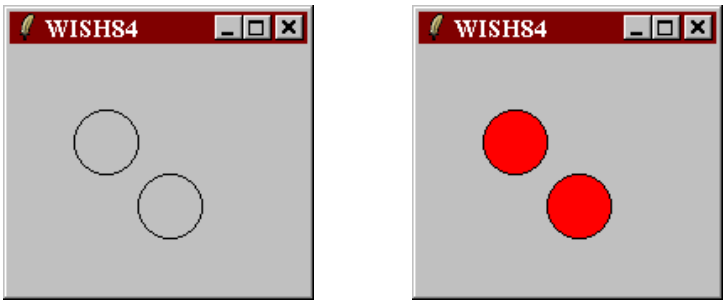
```
canvas .c
pack .c
.c create oval 1c 1c 4c 4c -fill red
```



Die Möglichkeiten, die Canvas bietet werden erst dadurch richtig interessant, dass man Objekten „Tags“ zuweisen kann. Dadurch ist es möglich mit `.c itemconfigure tag` (wobei `.c` der Name des Canvas ist) die Eigenschaften aller Objekte mit dem Tag `tag` mit nur einer Zeile verändern.

Beispiel:

```
canvas .c
pack .c
.c create oval 1c 1c 2c 2c -tag Kreis
.c create oval 2c 2c 3c 3c -tag Kreis
.c itemconfigure Kreis -fill red
```



6 Zusammenfassung

Zusammenfassend ist zu Tcl zu sagen, dass es sich hierbei um eine einfach zu erlernende Sprache handelt, die durch die wenigen vorgegebenen Einschränkungen sehr flexibel ist. Eigentlich besteht Tcl nur aus dem „Kommando-Konzept“. Alles weitere liegt in der Hand des Programmierers, wobei es natürlich eine große Anzahl bereits vordefinierter Kommandos gibt. Diese vordefinierten Kommandos sind aber von selbstdefinierten Kommandos durch nichts zu unterscheiden. Ein weiterer zentraler Punkt ist die Existenz von nur einem Datentypen, nämlich String. Diese Tatsache macht Quoting und Substitution nötig, welche sehr mächtige Instrumente zur Strukturierung von Kommandoaufrufen sind, aber schnell dazu führen das Skripte sehr unübersichtlich werden („quoting hell“). Insgesamt ist also festzuhalten, dass Tcl sehr flexibel und einfach zu erlernen ist, aber auch schnell sehr unübersichtlich wird.

Tk bietet zusätzlich die Möglichkeit grafische Benutzeroberflächen per Scripting zu erzeugen. Tk operiert in der Sprache Tcl und bietet eine Menge vordefinierter Kommandos zur Erzeugung grafischer „Widgets“ an. Somit wird die Flexibilität und Einfachheit von Tcl auch in Tk fortgeführt. Vor allem durch die Möglichkeiten des Packer bietet Tk die Möglichkeit sehr schnell und mit wenig Aufwand grafische Benutzeroberflächen zu erzeugen. Natürlich sind hier aber auch die oben genannten Nachteile von Tcl spürbar.

Es sei hier noch erwähnt, dass erst durch Tk auch Tcl – fortan mit Tcl/Tk bezeichnet – größere Verbreitung fand.

7 Literaturverweise

[1] John K. Ousterhout *Tcl and the Tk Toolkit*, Addison-Wesley, 1994
[2] John K. Ousterhout *Tcl: An Embeddable Command Language*, 1990
[3] John K. Ousterhout *An X11 Toolkit Based on the Tcl Language*, 1991
[4] http://www.okstate.edu/cis_info/cis_manual/jcl_over.html
[5] http://rhols66.adsl.netsonic.fi/era/unix/shell.html
[6] http://www.tcl.tk